# Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados

Carlos Alberto Kamienski

# Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Carlos Alberto Kamienski e aprovada pela Comissão Julgadora.

Campinas, 17 de março de 1994.

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.

# Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados[1]

Carlos Alberto Kamienski [2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Arthur João Catto (Orientador)[3]
- Carlos Antônio Ruggiero (Suplente)[4]
- João Antônio Zuffo[5]
- Nelson de Castro Machado[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 Ciência da Computação pela Universidade Federal de Santa Catarina (UFSC)

[3] Professor do Departamento de Ciência da Computação — IMECC/UNICAMP. Presidente da Fundação Centro Tecnológico para Informática (CTI)

[4] Professor do Instituto de Física e Química de São Carlos — IFQSC/USP.

[5] Professor do Laboratório de Sistemas Integráveis (LSI) da Escola Politéctica/USP.

[6] Professor do Departamento de Ciência da Computação — IMECC/UNICAMP.

*À Nil e à Gabriela,*
*Com muito carinho.*

# Agradecimentos

Ao CNPq pelo apoio financeiro recebido.

Ao meu orientador Prof. Arthur João Catto pela enorme boa vontade em me orientar à distância e confiar no meu trabalho.

Ao amigo Marcos Cezar Visoli pela dedicação inesquecível.

A minha esposa e minha filha, pela compreensão pelas inúmeras noites em que as deixei sozinhas.

"Me disseram que sonhar era ingênuo, e daí ?
A nossa geração não quer sonhar.
Pois, que sonhe a que há de vir"
(Oswaldo Montenegro)

# Resumo

Esta tese apresenta um estudo detalhado sobre o armazenamento de resultados (dados utilizados no processamento) em uma arquitetura de fluxo de dados. Ele segue uma forte tendência atual no sentido de unir as melhores características dos modelos von Neumann e de fluxo de dados em uma arquitetura híbrida.

Propõe-se a arquitetura da MX, uma máquina de fluxo de dados que incorpora mecanismos de gerenciamento explícito de memória (memória compartilhada dividida em módulos entrelaçados) e execução seqüencial de instruções. Mostra-se que esta arquitetura constitui uma plataforma adequada para a realização de testes de desempenho no sistema de memória.

# Abstract

This thesis presents a detailed study on result (data used in processing) storage in a data flow architecture. It follows a strong current tendency towards hibrid architectures which incorporate the best characteristics of the von Neumann and data flow computational models.

The MX architecture is proposed, a data flow machine which incorporates mechanisms for the explicit management of a shared memory partioned in interleaved modules and for sequential instruction execution. It is shown that such an architecture constitutes an adequate platform for performance testing of the memory system.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Os avanços tecnológicos ocorridos durante as últimas duas décadas tornaram possível construir eficientemente e a um baixo custo componentes de hardware poderosos, em especial, processadores e módulos de memória. No entanto, controlar o paralelismo potencial de um grande número de componentes cooperando para resolver problemas complexos é ainda um grande desafio aos pesquisadores.

A maioria dos esforços para construção de computadores altamente paralelos têm se concentrado em estender o modelo seqüencial dirigido por controle, acrescentando múltiplos processadores e proporcionando mecanismos de comunicação e sincronização entre eles. Esta abordagem representa a crença geral de que o maior problema com multiprocessamento está relacionado com o software. Em alguns casos os compiladores ou os programadores, devem dividir as aplicações em tarefas que podem executar em paralelo, usando primitivas de sincronização para assegurar um comportamento determinístico. Embora exista realmente um problema de software, faz-se necessária uma mudança fundamental na arquitetura do processador, antes que se observe avanços significativos no uso de multiprocessadores. A natureza dessa mudança é a integração profunda de sincronização no conjunto de instruções do processador, que deve constituir, na realidade, uma "linguagem de máquina paralela".

O conjunto de instruções de uma máquina de fluxo de dados [6] constitui tal linguagem de máquina paralela, na qual uma instrução é executada somente quando os seus operandos estão disponíveis. Assim, uma sincronização de baixo nível é realizada para cada instrução, na mesma taxa em que as instruções são encaminhadas para a execução. Para o compilador, torna-se fácil utilizar esta sincronização de granularidade fina para produzir um código altamente paralelo, cujos resultados são, no entanto, determinísticos, embora independentes da ordem

de execução. Apesar de ser consensual que fluxo de dados expõe a máxima quantidade de paralelismo possível num programa, em relação à eficiência dos seus mecanismos de execução as opiniões são muito divergentes.

Este trabalho diz respeito ao armazenamento de resultados (dados utilizados no processamento) em uma arquitetura de fluxo de dados. Ele investe na idéia de que é possível utilizar mecanismos de execução von Neumann em arquiteturas de fluxo de dados, aumentando sua eficiência sem perder a elegância do modelo. O trabalho faz parte das pesquisas realizadas pelo grupo de fluxo de dados do DCC/IMECC/Unicamp, que já possui trabalhos desenvolvidos nas áreas de estruturas de dados [69], geração de código [19, 84] e simuladores e modelos analíticos [76].

Os objetivos deste trabalho, em ordem crescente de importância, são:

- Fazer uma análise crítica das arquiteturas von Neumann e de fluxo de dados salientando os pontos positivos e negativos de ambas as abordagens para processamento paralelo. Estas informações são básicas para fundamentar novas propostas de arquiteturas híbridas von Neumann/fluxo de dados, que procuram reunir as melhores características dos dois modelos.

- Propor a arquitetura - aqui denominada MX -, de uma máquina capaz de executar o modelo de fluxo de dados dinâmico, incorporando mecanismos von Neumann tradicionais, como o gerenciamento de memória e a execução de blocos seqüenciais de código.

- Demonstrar que uma máquina de fluxo de dados pode utilizar o modelo de multiprocessamento de memória compartilhada sem penalizações no seu desempenho. Isto significa que o sistema de memória da MX deve ser capaz de fornecer dados a uma taxa muito alta, de modo a sua existência ser transparente ao resto do sistema.

Em particular, maior atenção é dedicada ao último item, e, em função dele, foram organizados os capítulos subseqüentes. O Capítulo 2 apresenta os conceitos básicos de arquitetura relacionada com processamento paralelo, enfatizando os problemas encontrados nos sistemas de memória. Além disso, introduz o conceito de arquitetura de fluxo de dados e a Máquina de Fluxo de Dados de Manchester (MFDM), que serviu de base para esta pesquisa.

O Capítulo 3 apresenta arquiteturas híbridas, que reúnem as melhores características dos modelos von Neumann e de fluxo de dados. A MX pode ser vista como uma arquitetura híbrida, que evoluiu do modelo de fluxo de dados em

direção a um aproveitamento mais eficiente dos recursos de hardware e melhor controle sobre a execução das instruções.

A arquitetura da MX é descrita no Capítulo 4. Seu alto desempenho potencial é garantido pelos seus princípios básicos de funcionamento, definidos após minucioso estudo da arquitetura da MFDM e das tendências atuais das pesquisas na área de arquiteturas híbridas.

No Capítulo 5 é feita uma avaliação da arquitetura da MX. Inicialmente é analisado o sistema de memória no campo teórico, sendo os argumentos apresentados um bom indicativo de que o sistema de memória projetado é capaz de suprir as necessidades da máquina. Além disso, são apresentados um programa exemplo e resultados de simulação bastante encorajadores sobre o desempenho da MX.

O Capítulo 6 encerra o trabalho apresentando algumas considerações e sugestões para futuras pesquisas.

# Capítulo 2

# Conceitos Básicos

A necessidade de sistemas computacionais com capacidade para executar diversas atividades em paralelo está se intensificando conforme crescem as demandas da nossa sociedade pela resolução de problemas mais complexos e por serviços mais sofisticados. Embora a tecnologia para produzir máquinas mais velozes continue a avançar, a utilização tradicional dessas máquinas não é capaz de satisfazer tais demandas. O que é necessário é a exploração do paralelismo, e para isso é indispensável a disponibilidade de máquinas com capacidade de processamento paralelo, isto é, máquinas com múltiplos elementos de processamento que executem simultaneamente.

As aplicações capazes de oferecer paralelismo suficiente para absorver o poder de processamento dessas máquinas, mantendo os elementos de processamento ocupados simultaneamente, podem ser distribuídas em duas categorias: processamento numérico e processamento simbólico [3]. Na primeira categoria encontram-se as aplicações mais tradicionais, que ofereceram as primeiras motivações para a pesquisa em processamento paralelo. Entre elas, podem-se citar aplicações científicas, de engenharia, em recursos energéticos, biológicas e ligadas à medicina. As aplicações da segunda categoria são mais recentes, incluindo sistemas de banco de dados e sistemas envolvendo inteligência artificial.

Nas seções subseqüentes, pretende-se apresentar questões básicas de arquitetura relacionadas com processamento paralelo, para em seguida, introduzir o conceito de arquiteturas de fluxo de dados e a Máquina de Fluxo de Dados de Manchester.

## 2.1 Arquitetura von Neumann

A arquitetura da maior parte dos computadores existentes atualmente é baseada em um modelo puramente seqüencial, proposto por von Neumann há mais de 40 anos, composto basicamente de três partes [14]:

- uma unidade central de processamento (UCP), comumente chamada processador;
- um local para armazenamento de informações, conhecido como memória;
- um canal de comunicação entre o processador e a memória, capaz de transmitir um único dado por vez, por este motivo chamado de "gargalo de von Neumann".

O processamento é controlado pelo "contador de programa" (PC), um registrador de propósito genérico embutido no processador, que mantém o endereço da próxima instrução a ser executada. O controle rígido que o PC exerce sobre a execução dos programas impõe a característica seqüencial dos computadores von Neumann.

A fim de projetar um computador que possa executar tarefas em paralelo, requerem-se alterações estruturais, dando origem a novos conceitos e configurações para o processador, a memória e o canal de comunicação. Deste último são derivadas as redes de interconexão.

## 2.2 Processadores

Processadores com capacidade de execução simultânea de múltiplas instruções são usualmente classificados em duas categorias [31]:

- SIMD (Single Instruction Multiple Data): possuem uma única linha de controle, aplicando uma mesma instrução a múltiplos dados;
- MIMD (Multiple Instruction Multiple Data): possuem várias linhas de controle, cada uma atuando sobre seus próprios dados.

Os processadores pertencentes à primeira categoria são ditos síncronos e compreendem os processadores vetoriais e os matriciais[1]. A maioria dos supercomputadores existentes atualmente baseia-se em processamento vetorial, por seu alto

[1] array processors

desempenho e relativa facilidade de implementação, em comparação com outros processadores paralelos. Isto é devido ao "paralelismo temporal" que é explorado nesses processadores, decorrente da sobreposição da execução de certas fases de processamento, um mecanismo conhecido como *pipelining*. Esse mecanismo é também um dos pontos fundamentais das arquiteturas RISC [63].

A segunda categoria compreende os processadores assíncronos, chamados multiprocessadores. Estes são, na maior parte dos casos, formados por uma coleção de processadores seqüenciais, que seguem o modelo von Neumann. Sua necessidade se torna aparente quando se trata de problemas que não podem ser facilmente organizados em operações repetitivas sobre dados uniformemente estruturados [80]. Na prática, grande parte dos problemas são freqüentemente não estruturados e com padrões de endereçamento imprevisíveis, o que inviabiliza a utilização de processadores SIMD. Processadores MIMD exploram "paralelismo espacial", ou seja, cada processador é independente, sendo capaz de executar seu próprio programa.

Multiprocessadores são divididos em duas classes, de acordo com o modo de acesso à memória [3, 47]: no modelo de memória compartilhada, todos os processadores dividem um espaço de endereçamento comum; no modelo de memória distribuída, ou de passagem de mensagens, cada processador possui sua própria memória privada.

Não existe ainda um consenso sobre o melhor modelo de multiprocessador a ser adotado. Tradicionalmente, os multiprocessadores possuíam memória compartilhada, o que introduz o problema de conflitos, como será visto na Seção 2.3. Multiprocessadores com memória distribuída surgiram justamente como uma solução para este problema. No entanto, eles introduzem outras questões ainda não resolvidas adequadamente.

Projetistas de hardware e software têm opiniões divergentes sobre a escolha entre memória compartilhada ou distribuída. Do ponto de vista do software, a mais atrativa para a programação é aquela que proporciona um espaço de endereçamento único, com variáveis compartilhadas. Além disso, a troca de mensagens pode ser simulada, através do uso de *buffers* na memória compartilhada. A programação em memória distribuída é mais complexa e se apóia na utilização de primitivas *send/receive* para a troca de mensagens.

Do ponto de vista do hardware, proporcionar acesso simultâneo à memória compartilhada, com latência razoável (vide Seção 2.3) e custo aceitável ainda é um desafio. Multiprocessadores de memória distribuída apresentam um custo mais baixo e uma simplicidade muito maior. Porém, eles reduzem a mobilidade dos processos, que são vinculados aos processadores onde seus dados estão localizados.

## 2.3 Memória

Uma das questões mais sérias envolvidas no projeto de sistemas computacionais de alto desempenho está relacionada com a latência de memória. Latência é o tempo transcorrido entre o envio de uma requisição à memória por um processador e o recebimento pelo processador do item de dado associado [10].

### 2.3.1 Latência em processadores seqüenciais

Em processadores seqüenciais tradicionalmente têm sido usadas com sucesso memórias *cache* para diminuir o tempo durante o qual o processador fica ocioso esperando o retorno da informação desejada. Este é um mecanismo que torna menor o tempo de acesso da memória, pelo menos na maior parte dos casos. Outra técnica utilizada para superar este problema é o *pipelining*, de modo que, enquanto uma instrução faz acesso à memória, outras estão sendo decodificadas e executadas. Esta leitura antecipada das instruções somente é possível porque os programas são implicitamente seqüenciais. Desde que o esvaziamento de estágios do *pipeline* devido à ocorrência de instruções de desvio ou dependências de dados não seja muito freqüente, na maior parte dos acessos à memória o processador não ficará ocioso.

Arquiteturas LOAD/STORE minimizam seus problemas com latência dividindo seu conjunto de instruções em duas classes de instruções distintas. Numa classe estão as instruções que movem dados inalterados entre a memória e registradores internos. Na outra estão as instruções que operam sobre esses dados nos registradores, as quais podem fazer acesso à memória. Esta distinção rígida simplifica o escalonamento das instruções, uma vez que é trivial verificar se uma instrução necessita ou não de uma referência à memória. Mais do que isto, o sistema de memória e a ALU podem ser vistos como *pipelines* independentes, o que aliado a um grande número de registradores internos que podem ser reservados antecipadamente para receber os dados de determinada instrução, torna possível a redução dos custos da latência.

Entre as máquinas que seguem esta filosofia estão as máquina RISC e alguns supercomputadores como o Cray-1 [26]. Arquiteturas LOAD/STORE geralmente utilizam outras técnicas para a resolução do problema da latência, como memória *cache* e *pipelining*. Desse modo, conseguem assumir que referências à memória levam uma quantidade fixa de tempo (um ciclo, na maioria das máquinas RISC), ou que a quantidade de tempo é variável, mas perfeitamente previsível (como no Cray-1).

### 2.3.2 Latência em processadores paralelos

Em ambientes de processamento paralelo esta questão é radicalmente mais complexa. Como múltiplos dados devem ser processados simultaneamente, existe a necessidade de que eles sejam trazidos da memória de uma única vez. No esquema convencional de memória isto não é factível, devido à impossibilidade de um componente físico de memória responder a mais do que uma requisição ao mesmo tempo. Esta limitação é imposta basicamente pela composição lógica da memória em um único módulo e pela impossibilidade da transmissão simultânea de dados pelo canal de comunicação.

Uma solução natural para este problema é a partição da memória em vários módulos, com os endereços distribuídos entre eles, de modo que, quanto mais módulos houver, mais acessos simultâneos são permitidos. Este esquema é chamado entrelaçamento[2] [47]. Existem dois métodos básicos para distribuir os endereços entre os módulos de memória. Quando endereços consecutivos são localizados no mesmo módulo tem-se um entrelaçamento de ordem alta. Entrelaçamentos de ordem baixa ocorrem quando endereços consecutivos localizam-se em módulos distintos. O segundo método é o mais adequado a processadores paralelos e sua utilização é bastante eficaz, principalmente no âmbito de processadores vetoriais e matriciais.

Multiprocessadores de memória distribuída levam grande vantagem quanto à latência de memória, pois o porte deste problema é praticamente reduzido ao dos processadores seqüenciais. Multiprocessadores de memória compartilhada, ao contrário, têm de conviver com problemas muito maiores devido à latência. Embora alguns pesquisadores tentem amenizar o impacto negativo da latência em multiprocessadores de memória compartilhada [66, 78], as seguintes constatações podem ser feitas [10, 26]:

- a existência de muitos processadores realizando acessos assíncronos e imprevisíveis ao mesmo módulo de memória pode gerar contenção de memória;
- o tempo para buscar um operando pode não ser constante, porque alguns módulos de memória podem estar "mais perto" do que outros na organização física da máquina;
- não se pode prever o pior caso para o tempo de latência se o requisito da escalabilidade for desejado para a máquina;

[2] interleaving

- a presença de uma rede de interconexão introduz um hiato inevitável entre a requisição de um acesso à memória por um processador e o recebimento da resposta correspondente[3].

Em geral, pode-se concluir que quanto maior o tamanho de um multiprocessador (ou seja, quanto maior o número de processadores e módulos de memória) maior será a latência média.

### 2.3.3 Resolução de latência em processadores paralelos

A resolução dos problemas decorrentes da latência de memória em processadores paralelos é tratada com base em duas possíveis classes de aplicações-alvo: processamento vetorial e processamento escalar. Em processamento vetorial a largura de banda da memória[4] é mais importante que a latência, porque esta pode ser escondida pela sobreposição de operações de memória através de entrelaçamento. Neste caso são utilizados métodos que visam aumentar o desempenho da memória principal da máquina com técnicas específicas para resolver conflitos nos acessos à memória entrelaçada, para processadores vetoriais, matriciais e multiprocessadores. Estas questões são discutidas com mais detalhes no Capítulo 5.

Em processamento escalar são utilizados métodos para reduzir e tolerar a latência, entre os quais se destacam: memórias *cache* coerentes [79], relaxamento da consistência de memória, busca antecipada e múltiplos contextos [87]. Resultados de simulação demonstraram que combinações das várias técnicas geralmente obtêm melhor desempenho do que cada uma isoladamente [40].

## 2.4 Redes de interconexão

Para que os processadores possam cooperar na execução de um problema, deve haver algum esquema de conexão entre eles. Nos multiprocessadores com memória distribuída, os processadores comunicam-se diretamente, havendo a necessidade de uma rede de interconexão entre eles. Nos multiprocessadores com memória compartilhada, a comunicação é indireta, através da memória. Nesse caso, a rede de interconexão é voltada à comunicação entre os processadores e a memória, eliminando-se a ligação direta eles.

Uma rede de interconexão pode ser avaliada em termos da velocidade com a qual consegue transmitir dados confiavelmente a um custo razoável. Os critérios

[3] Os problemas relacionados a redes de interconexão serão discutidos na próxima seção.

[4] *Memory bandwidth* - número de palavras a que o sistema de memória pode ter acesso por segundo.

de desempenho usuais para avaliar uma rede conectando elementos de processamento são [3]:

- Latência: o tempo para transmissão de uma mensagem;
- Largura de banda: a quantidade de mensagens que a rede consegue manusear por segundo;
- Conectividade: quantos vizinhos imediatos tem cada elemento de processamento;
- Custo relativo: quanto representa o custo da rede no custo total do hardware;
- Confiabilidade: a existência de caminhos redundantes e outras facilidades que proporcionam segurança em caso de danos à rede.

As redes de interconexão podem ser divididas, quanto à sua topologia, em estáticas e dinâmicas [29]. Redes estáticas não mudam depois que a máquina é construída, e são convenientes para problemas cujos padrões de comunicação podem ser antecipados confiavelmente. Redes dinâmicas, apesar de mais caras, são adequadas para uma classe maior de problemas, por possuírem a capacidade de se alterar conforme as necessidades de comunicação.

Existem várias configurações possíveis para redes de interconexão com topologia estática. A mais potente de todas é a conexão ponto-a-ponto, onde cada um dos nós é conectado a todos os outros. Porém esta topologia é proibitivamente cara para configurações com mais do que alguns processadores. Outras ligações possíveis para redes com topologia estática são: linear, matricial, em anel, em estrela e em hipercubo. Avanços recentes em tecnologia de VLSI tornaram viável técnica e economicamente este tipo de rede, principalmente no âmbito de multiprocessadores com memória distribuída [65].

Redes de interconexão com topologia dinâmica são mais utilizadas em multiprocessadores com memória compartilhada, onde podem ser utilizadas para a resolução de uma classe muito maior de problemas. Uma rede baseada em barramento compartilhado é a menos complexa, a mais barata e também a mais utilizada. O barramento não permite mais do que uma transferência entre os processadores e a memória por vez. A largura de banda disponível para cada processador é inversamente proporcional ao número de processadores. Por outro lado, uma rede *crossbar* suporta todas as conexões possíveis com a memória simultaneamente. Infelizmente, o custo de tal rede é $O(N^2)$ para conectar N entradas e N saídas. Para uma máquina com um número grande de processadores

e memórias, tanto o barramento quanto o *crossbar* são inviáveis; o primeiro pelo baixo desempenho e o segundo pelo custo elevado.

Em termos de custo e desempenho, redes multiestágio e redes de múltiplos barramentos constituem um meio-termo razoável entre o barramento compartilhado e o *crossbar* [16]. Redes multiestágio permitem um rico subconjunto de ligações entre processadores e módulos de memória, ao mesmo tempo que reduzem o custo do hardware para $O(NlogN)$. As redes de múltiplos barramentos são uma extensão natural do barramento compartilhado, oferecendo maior largura de banda e tolerância a falhas a um custo razoável. Por este motivo, têm recebido muita atenção dos pesquisadores nos últimos anos.

## 2.5 Arquiteturas de fluxo de dados

A semântica do modelo computacional de von Neumann é imperativa, ou seja, existe um controle explícito do programa sobre a ordem da execução das instruções. Além disso, existem estados aparentes ao programador, em decorrência da associação entre nomes (variáveis) no programa e posições de armazenamento. Como conseqüência da combinação destas duas propriedades, podem ocorrer efeitos colaterais na execução das instruções.

Arquiteturas de fluxo de dados são multiprocessadores baseados no modelo de fluxo de dados, onde a semântica das operações sobre os dados é funcional. Não existem efeitos colaterais, pois a única dependência que se estabelece está entre dado e operação. Uma operação comporta-se como uma função: toma argumentos e produz resultados. Os argumentos permanecem inalterados. O controle sobre a execução das operações é implícito, sendo estabelecido a partir das dependências existentes entre as próprias operações. As principais características do modelo de fluxo de dados são [82, 83]:

- uma instrução está pronta para executar no momento em que todos os seus operandos estiverem disponíveis;
- resultados intermediários ou finais são passados diretamente entre instruções;
- não existe o conceito de memória de dados, de acordo com a noção tradicional de variável;
- a seqüência de execução das instruções é determinada somente pelas dependências de dados, abrindo um potencial muito grande para processamento paralelo.

### 2.5.1 Grafos de fluxo de dados

Grafos de fluxo de dados [28] são grafos orientados, onde os vértices correspondem às operações e as arestas aos caminhos percorridos pelos dados. Os dados são representados como fichas, que, de acordo com a perspectiva, podem ser vistos como argumentos ou resultados. Um grafo de fluxo de dados é um programa em nível de máquina, que pode ser executado diretamente, com base na equivalência entre as operações (vértices) do grafo e as instruções da máquina. Cada instrução contém um código de operação e uma lista de endereços de destino e cada ficha é formada por dois campos: o valor (dado) e o endereço da aresta de destino. O processo computacional em uma máquina de fluxo de dados pode ser visto como o fluxo das fichas pelo grafo do programa correspondente.

O disparo de uma instrução, ou vértice, do grafo de fluxo de dados ocorre quando uma ficha está disponível em cada uma de suas arestas de entrada. O ciclo básico de execução de instruções em um grafo de fluxo de dados é [5]:

(a) detectar quando uma operação está habilitada, o que compreende sincronizar as fichas de entrada e consumí-las;

(b) determinar a operação a ser realizada;

(c) computar os resultados;

(d) gerar fichas de resultado para cada aresta de saída.

O alto grau de paralelismo disponível no modelo de fluxo de dados pode ser observado nesta forma gráfica (em duas dimensões) de representar um programa: instruções que podem ser executadas simultaneamente são dispostas lado a lado, e instruções que devem ser executadas em seqüência, umas sobre as outras. Grafos de fluxo de dados representam o programa como uma ordem parcial de dependências essenciais, e instruções são escalonadas dinamicamente com base na disponibilidade dos dados. Isto é, um programa para uma máquina de fluxo de dados não especifica completamente a ordem real da execução das instruções, baseando-se em mecanismos de execução de baixo nível (p.ex.: sincronização de operandos), para dinamicamente tomar uma particular ordem de execução. Desse modo, a computação é assíncrona, mas determinística, ou seja, independente da ordem de execução escolhida, o resultado será correto.

### 2.5.2 Linguagens de fluxo de dados

Grafos de fluxo de dados formam uma linguagem paralela de máquina poderosa, não sendo entretanto adequados como interface para o desenvolvimento de siste-

mas complexos. Desse modo, existe a necessidade de prover uma linguagem de programação de alto nível para computadores de fluxo de dados a partir da qual os grafos possam ser gerados. Embora qualquer linguagem de alto nível possa ser utilizada, é desejável que ela possua características das linguagens funcionais, a fim de levar vantagem do paralelismo implícito do modelo. Algumas propriedades úteis para linguagens de fluxo de dados são [1]:

- *Localidade de efeito*: esta propriedade pode ser obtida se instruções não tiverem dependências de dados de longo alcance, ou seja, o efeito das instruções deve ter uma localidade bem definida.
- *Ausência de efeitos colaterais*: esta propriedade é necessária para assegurar que as dependências de dados são consistentes com as restrições de seqüenciamento das instruções. Linguagens de fluxo de dados baseiam-se na aplicação de funções sobre dados, impedindo a ocorrência de efeitos colaterais.
- *Regra de atribuição única*: uma variável pode aparecer somente uma vez no lado esquerdo de uma atribuição dentro da área de programa em que ela está ativa. Esta propriedade facilita a detecção de paralelismo em um programa.
- *Desdobramento de laços*: em uma computação iterativa, os vários ciclos devem ser desdobrados, se não possuírem dependências de dados entre si, para que sejam executados simultaneamente.

### 2.5.3 Modelo estático e dinâmico

O mecanismo de armazenamento de fichas é a característica mais importante das arquiteturas de fluxo de dados. O modelo de fluxos de dados assume filas FIFO com capacidade ilimitada nas arestas e comportamento FIFO nos vértices, mas uma implementação a este nível torna-se muito difícil. Duas abordagens alternativas têm sido estudadas extensivamente. A primeira chama-se fluxo de dados estático, que prevê uma quantidade fixa de armazenamento por aresta. A outra abordagem chama-se fluxo de dados dinâmico, que provê alocação dinâmica de armazenamento de fichas, e assume que as fichas carregam rótulos para indicar sua posição lógica nas arestas.

As máquinas que se baseiam no modelo de fluxo de dados estático geralmente impõem a restrição de que, em qualquer instante, pode haver no máximo uma ficha em cada aresta. Esta restrição pode ser incorporada ao modelo estendendo

a regra de disparo a fim de tornar necessário que todas as arestas de saída de um vértice estejam vazias para que ele esteja habilitado a executar. Dessa forma, o armazenamento para fichas pode ser alocado antecipadamente, uma vez que o número de arestas é fixo para um determinado grafo. O formato de instrução básico deve ser expandido para incluir um campo para cada operando. Os campos têm sinalizadores de presença para indicar se um valor já foi armazenado, de modo que quando uma ficha é armazenada, é fácil determinar se as outras entradas estão todas presentes. A restrição de uma ficha por aresta pode ser implementada através de arestas de reconhecimento. A implementação do modelo de fluxo de dados estático é relativamente simples, mas em contrapartida, impede a utilização de recursividade e limita o desdobramento de laços em sua total generalidade. Isto significa que o potencial de paralelismo existente em um grafo de fluxo de dados não pode ser plenamente explorado.

Quanto à exploração de paralelismo, as máquinas que se baseiam no modelo dinâmico de fluxo de dados aproximam-se bastante do modelo de fluxo de dados abstrato. Através da incorporação de rótulos às fichas, que distinguem os contextos de várias ativações de um determinado grafo, várias fichas podem compartilhar uma mesma aresta, possibilitando a execução concorrente de múltiplas instâncias do código. Em fluxo de dados dinâmico realizad-se uma associação entre grafos, chamados "blocos de código", e funções definidas pelo usuário. Cada grafo ou é acíclico ou é um laço simples. A atribuição de um rótulo exclusivo a cada particular ativação de um bloco de código permite então o aproveitamento correto do paralelismo existente em chamadas de função concorrentes, iterações e recursividade. Isto é feito sem comprometer as características de controle distribuído e assincronismo na execução das operações.

A inclusão de um campo destinado ao rótulo nas fichas provoca alterações nas regras de disparo dos vértices (instruções), que passam a ser:

- Um vértice está habilitado se, e somente se, existirem fichas com o mesmo rótulo em todas as suas arestas de entrada.
- Qualquer conjunto de vértices habilitados pode ser disparado simultaneamente.
- Disparar um vértice implica na remoção de um conjunto completo de fichas de mesmo rótulo das arestas de entrada e na produção de um conjunto coerente de fichas-resultado, onde o conteúdo dos campos de valor e rótulo é determinado pelo tipo de operação executada.

Ao contrário do modelo estático, o modelo permite o disparo de vértices

mesmo quando existem fichas nas suas arestas de saída. Ao mesmo tempo, perde-se a simplicidade do mecanismo de sincronização de operandos do modelo estático. As implementações do modelo dinâmico tornam imprescindível a separação entre código e dados, sendo o armazenamento e sincronização de operandos usualmente realizados através de memória associativa. Problemas decorrentes dessa abordagem são relatados nos Capítulos 3 e 4.

## 2.6 A Máquina de Fluxo de Dados de Manchester

A Máquina de Fluxo de Dados de Manchester (MFDM) [42, 86] é o fruto de um projeto que se iniciou em meados dos anos 70 na Universidade de Manchester e culminou com a construção de um protótipo, que esteve operacional entre 1981 e 1989. A máquina baseia-se no modelo de fluxo de dados dinâmico, associando a cada ficha um rótulo para permitir múltiplas ativações simultâneas de um mesmo bloco de código. Além disso, ela é assíncrona pois a ordem de execução das instruções não é garantida, devido à regra do próprio modelo que torna qualquer subconjunto de instruções habilitadas um candidato potencial para ser executado. O paralelismo no sistema é de granularidade fina, sendo explorado ao nível de instruções.

O formato da ficha na MFDM pode ser descrito como:

(dado, rótulo, destino)

O tamanho máximo dos dados que a MFDM suporta diretamente é de 32 bits. As diferentes instâncias de um bloco de código são identificadas através do rótulo, que é formado por dois campos: *nome de ativação* e *índice*. O nome de ativação é um identificador único no sistema, que separa os contextos das várias instâncias de um bloco de código. O índice é utilizado para distinguir elementos pertencentes a uma mesma estrutura de dados.

Uma diagrama de blocos da MFDM é mostrado na Figura 2.1. A estrutura básica é um anel onde as unidades estão conectadas a um computador hospedeiro, que proporciona funções de entrada e saída, uma vez que o sistema não possui capacidade de execução autônoma. As unidades operam independentemente, formando um grande *pipeline*. As fichas são encapsuladas em pacotes de dados, que circulam no anel sempre no mesmo sentido. Cada unidade no *pipeline* é internamente síncrona, mas comunica-se com as outras unidades através de um protocolo assíncrono. O paralelismo nesta estrutura está na Unidade Funcional, onde múltiplos elementos de processamentos são necessários para suportar a taxa

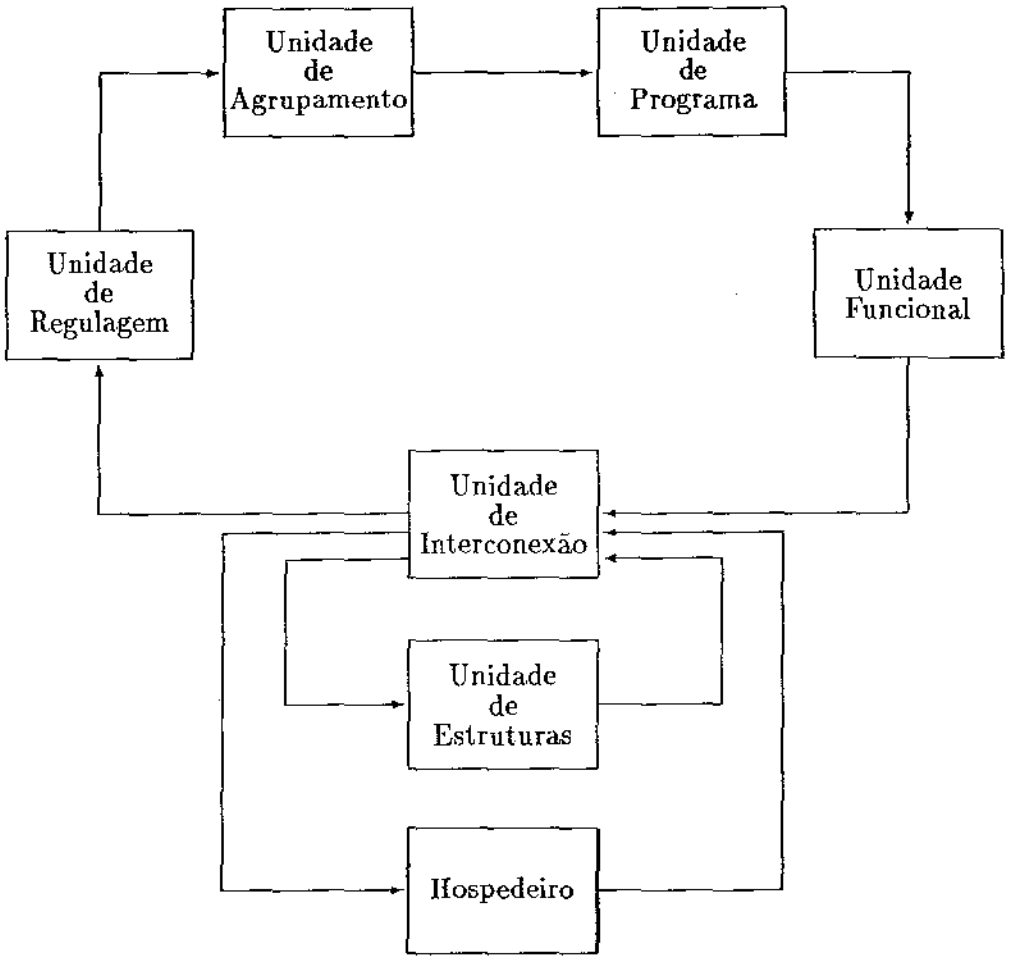



Figura 2.1: A Máquina de Fluxo de Dados de Manchester (MFDM)

de produção de dados no anel. Adicionalmente, existe o paralelismo temporal no próprio anel, uma vez que as operações em cada unidade são sobrepostas. A seguir, são apresentadas as unidades que compõem a MFDM.

### 2.6.1 Unidade de Regulagem

A Unidade de Regulagem (*Token Queue*) é uma fila FIFO circular que abranda as flutuações das taxas de geração e consumo de fichas. Ela armazena temporariamente as fichas recebidas da Unidade Funcional de modo a prover uma taxa de alimentação mais equilibrada à Unidade de Agrupamento durante a execução

de um programa.

### 2.6.2 Unidade de Agrupamento

A Unidade de Agrupamento (*Matching Unit*) sincroniza as fichas destinadas a uma mesma instrução. Estas são no máximo duas e contêm o mesmo rótulo e o mesmo endereço de destino. Ao ingressar na Unidade de Agrupamento uma ficha pode receber dois tipos de tratamento: a) ela é destinada a uma instrução com apenas uma entrada e passa livremente por este módulo; b) ela é destinada a uma instrução com 2 entradas. Neste último caso é feita uma tentativa de encontrar a sua parceira. Caso a parceira ainda não tenha chegado, a ficha é armazenada. Caso contrário a parceira é retirada e as duas fichas colocadas num pacote de grupo com o seguinte formato:

(dado1, dado2, rótulo, destino)

A sincronização de fichas é uma parte crítica da operação da máquina. Ela deve ser associativa por natureza, além de permitir o armazenamento de uma grande quantidade de fichas que não encontram suas parceiras. Uma vez que memórias totalmente associativas ainda apresentam um custo muito elevado, a Unidade de Agrupamento foi implementada de forma pseudo-associativa, utilizando os campo de rótulo e destino das fichas como chave de acesso a bancos de memória convencionais, recorrendo a um mecanismo de *hashing* implementado em hardware.

### 2.6.3 Unidade de Programa

Antes do início da execução, as instruções do programa são carregadas na Unidade de Programa (*Node Store*). Durante a execução, a unidade realiza a busca de instruções a partir das fichas agrupadas, e também daquelas destinadas a instruções de apenas uma entrada, enviadas pela Unidade de Agrupamento. O acesso às instruções é definido pelo campo de destino, que é dividido em duas partes: uma identificando a base do segmento e outra um deslocamento.

As instruções podem assumir uma das seguintes formas:

(código de operação, destino), ou
(código de operação, destino1, destino2), ou
(código de operação, destino, literal)

O pacote resultante está pronto para a execução e é enviado à Unidade Funcional na forma:

(dado1, dado2, código de operação, rótulo, destino1, destino2)

### 2.6.4 Unidade Funcional

A Unidade Funcional (*Functional Unit*) é composta por um conjunto de processadores dispostos em paralelo, que realizam operações lógicas, aritméticas e de controle sobre as fichas. Um pequeno número de instruções é executada por um pré-processador especializado. A maioria das instruções é encaminhada a um sistema de distribuição, responsável pela seleção de um processador livre. Um sistema de arbitragem encarrega-se de encaminhar de volta ao anel as fichas produzidas como resultado pelos diversos processadores.

Por sua natureza, uma máquina experimental requer um projeto com a máxima flexibilidade possível. Por esta razão os processadores da Unidade Funcional são microprogramáveis, de modo que alterações no conjunto de instruções possam ser efetuadas sem maiores complicações.

### 2.6.5 Unidade de Interconexão

A Unidade de Interconexão (*Switch*) efetua a comunicação do anel com o mundo externo, que pode compreender um computador hospedeiro, a Unidade de Estruturas, e outros anéis em uma arquitetura multianel [41]. Uma vez que possui várias entradas e saídas independentes, a Unidade de Interconexão opera a uma velocidade maior que a das outras unidades, para atender às necessidades de comunicação sem causar atrasos desnecessários. Quando há fichas chegando simultaneamente, a preferência é dada àquelas oriundas da Unidade Funcional.

### 2.6.6 Unidade de Estruturas

A Unidade de Estruturas (*Structure Store*) [71, 51] comunica-se com o anel através da Unidade de Interconexão e foi desenvolvida para melhorar o desempenho de programas envolvendo estruturas de dados. No protótipo inicial da MFDM, todas as estruturas eram armazenadas na Unidade de Agrupamento usando funções de agrupamento especiais [20] que produzem fichas *grudentas*. Estas, quando agrupadas, são copiadas mas não extraídas. Entretanto, esta abordagem revelou-se ineficiente por três razões principais:

- O número elevado de instruções intermediárias necessárias para o acesso a elementos de uma estrutura reduz o desempenho e aumenta a latência de leitura.
- O espaço disponível para armazenamento fica limitado ao tamanho da Unidade de Agrupamento que também é utilizada por escalares.
- O custo de armazenamento é alto e o desempenho da Unidade de Agrupamento fica comprometido pelo excesso de movimento.

A Unidade de Estruturas da MFDM foi dirigida para o armazenamento de Estruturas-I [12] e isto repercutiu consideravelmente na sua organização e complexidade. Ela é formada por quatro partes: a Sub-unidade de Alocação, a Sub-unidade de Armazenamento, a Sub-unidade de Limpeza e a Sub-unidade de Leituras Antecipadas. As sub-unidades realizam tarefas independentes e podem operar concorrentemente.

A Sub-unidade de Alocação efetua a reserva de espaço para o armazenamento de estruturas na Sub-unidade de Armazenamento. A gerência de memória é realizada totalmente pelo hardware. Na Sub-unidade de Armazenamento são mantidos os elementos de uma estrutura de dados. Para suportar Estruturas-I, existem em cada endereço 2 bits que determinam se o elemento está presente e se ocorreram leituras antecipadas naquele endereço. O papel da Sub-unidade de Limpeza é remover os elementos de uma estrutura fora de uso e devolver o seu espaço à lista de espaço disponível na Sub-unidade de Alocação. Na Sub-unidade de Leituras Antecipadas são armazenadas as requisições de leitura de elementos que ainda não se encontram presentes na Sub-unidade de Armazenamento. Estas requisições pendentes são armazenadas em listas, sendo que cada lista está associada a um único endereço na Sub-unidade de Armazenamento.

## 2.7 Resumo

Neste capítulo foram apresentados os principais conceitos envolvendo as tecnologias utilizadas pelas máquinas com capacidade para processamento paralelo. Foi visto que a arquitetura básica von Neumann, composta de processador, memória e canal de comunicação, deve sofrer alterações a fim de que possa vencer as suas limitações com relação a processamento paralelo. Em especial, o sistema de memória deve receber uma grande atenção, pois pode facilmente tornar-se o maior gargalo de uma arquitetura paralela, seja ela SIMD ou MIMD. Devido à ação conjunta de vários fatores, essas máquina têm de conviver com problemas

muito sérios devido à latência de memória. A resolução desses problemas recebe tratamento diferenciado dependendo se o sistema computacional se destinar ao processamento escalar ou vetorial.

O modelo de fluxo de dados é apresentado como um candidato promissor para a construção de sistemas capazes de explorar um alto grau de paralelismo. Ele é baseado na disponibilidade dos dados, ou seja, uma operação somente é habilitada no momento em que todos os seus operandos estão presentes. Os programas em linguagem de máquina nesse caso podem ser representados por grafos de fluxo de dados, grafos orientados em que os vértices representam operações, e as arestas, os caminhos percorridos pelos dados. Os dados, por sua vez, são representados por fichas. Grafos de fluxo de dados podem ser gerados diretamente a partir de uma linguagem de programação, e, embora em princípio qualquer linguagem de alto nível possa ser utilizada, é desejável que ela possua características das linguagens funcionais.

Arquiteturas de fluxo de dados podem ser classificadas como estáticas ou dinâmicas. No modelo estático, somente é permitida uma ficha por aresta no grafo de fluxo de dados em qualquer momento, restringindo o paralelismo. No modelo dinâmico, múltiplas fichas podem ocupar simultaneamente uma mesma aresta, o que aumenta o paralelismo por permitir reentrância de código, mas também aumenta a complexidade e encarece a implementação.

Como base para o estudo de arquiteturas de fluxo de dados foi apresentada a Máquina de Fluxo de Dados de Manchester (MFDM). A MFDM baseia-se no modelo dinâmico, associando a cada ficha um rótulo, para melhor explorar o paralelismo. Sua estrutura básica é um anel, que interliga diversas unidades de propósito específico. Os detalhes da arquitetura descritos são importantes para orientar a leitura do Capítulo 3, mas formam, principalmente, uma base indispensável para a compreensão dos motivos que levaram ao projeto da MX, no Capítulo 4.

# Capítulo 3

# Arquiteturas Híbridas

Quais características as arquiteturas von Neumann podem tomar emprestadas das arquiteturas de fluxo de dados para tornarem-se mais adequadas ao processamento paralelo? E, por sua vez, quais as características que as arquiteturas de fluxo de dados podem emprestar das arquiteturas von Neumann para aumentar sua eficiência? A resposta para estas perguntas é a chave para a descoberta de uma nova classe de arquiteturas paralelas de alto desempenho que poderão revolucionar o mercado de supercomputadores nos próximos anos. Até o presente momento, os processadores vetoriais têm obtido maior sucesso neste sentido, mas isto somente ocorre porque as arquiteturas MIMD ainda não alcançaram seu objetivo mais importante, a escalabilidade. Um multiprocessador é dito escalável quando o ganho em desempenho é proporcional à quantidade de recursos de hardware adicionados [10].

Arquiteturas de fluxo de dados e von Neumann têm muitas informações importantes a trocar e pesquisadores das duas áreas concordam que a síntese das duas pode tornar viável a construção de máquinas com centenas, e talvez milhares, de processadores apresentando alto desempenho na execução de programas reais. Neste trabalho, arquiteturas resultantes da união dos modelos fluxo de dados e von Neumann serão chamadas simplesmente arquiteturas híbridas.

Este capítulo inicia com uma análise crítica das arquiteturas von Neumann e de fluxo de dados, para que se ganhe algum discernimento a respeito delas. Então são lançadas idéias para a união das melhores características das duas, em busca de um modelo híbrido. Para finalizar são apresentadas algumas propostas que podem ser enquadradas nesta nova classe de arquiteturas híbridas.

## 3.1 Crítica às arquiteturas von Neumann

É interessante notar que as lições aprendidas em 40 anos otimizando arquiteturas von Neumann não levam necessariamente a multiprocessadores. Técnicas de compilação associadas com o projeto de *pipelines*, utilizadas com sucesso em processadores seqüenciais, não se ajustam facilmente a máquinas com múltiplos focos de controle simultâneos e um comportamento assíncrono difícil de prever. Do mesmo modo, as linguagens imperativas de alto nível, que conseguem explorar eficientemente os recursos das máquinas seqüenciais, não conseguem suprir adequadamente as necessidades do processamento paralelo.

### 3.1.1 Duas questões fundamentais: latência e sincronização

A maioria dos projetos de máquinas paralelas disponíveis comercialmente hoje em dia é baseada em processadores von Neumann operando com memória compartilhada. As diferenças nas arquiteturas dessas máquinas em termos de velocidade do processador, organização de memória e sistemas de comunicação são significativas, mas todas elas utilizam processadores von Neumann relativamente convencionais. Essas máquinas refletem a crença geral de que a arquitetura do processador tem pouca importância no projeto de máquinas paralelas. Essa suposição não é verdadeira e isto pode ser comprovado com base em duas questões consideradas fundamentais: latência de memória e sincronização [10]. Esse argumento é baseado nas seguintes observações:

(a) A maioria dos processadores von Neumann fica ociosa durante longas referências à memória, que são absolutamente inevitáveis em uma máquina paralela.

(b) Atrasos por eventos de sincronização requerem freqüentes trocas de contexto para evitar a ociosidade do processador, que são caras em máquinas von Neumann.

A latência é associada diretamente à partição física do multiprocessador, enquanto que a sincronização é mais uma função de como os programas são decompostos. Estas questões não são apenas fundamentais, elas são fortemente relacionadas entre si. Uma possível solução para tolerar a latência é a troca de contexto, mas para isso é necessário um mecanismo altamente eficiente de sincronização, capaz de agrupar os processos bloqueados e os acessos pendentes à memória. Por outro lado, o tratamento eficiente de um evento de sincronização requer também uma troca de contexto, cujo custo é altamente dependente do

estado que o processador carrega. Aumentar o estado do processador, caracterizado principalmente pela quantidade de registradores, é uma técnica utilizada freqüentemente para reduzir o número de acessos à memória.

Algumas questões relacionadas ao aumento de desempenho do sistema de memória e técnicas para reduzir e tolerar o efeito da latência de memória já foram discutidas no Capítulo 2.

Sincronização é freqüentemente necessária para a comunicação entre processos e a eficiência dos mecanismos que realizam a sincronização determina a freqüência com que é viável os processadores estabelecerem comunicação. Os participantes de qualquer evento de sincronização necessitam de um motivo comum, um ponto de encontro, que pode ser um semáforo, um registrador, um rótulo de *buffer*, um nível de interrupção, etc. Uma máquina von Neumann em uma configuração multiprocessadora apresenta desempenho baixo porque não suporta sincronização eficiente a um baixo nível. Freqüentemente deseja-se utilizar processadores convencionais disponíveis comercialmente e as sincronizações são realizadas por interrupções, que são procedimentos caros por forçarem a ocorrência de uma troca de contexto. Desse modo, é necessário que a comunicação seja bastante infreqüente, o que exige processos de granularidade grossa e inibe parcela considerável do paralelismo potencial existente em um dado problema. Por outro lado, estes mecanismos de sincronização também não são apropriados para controlar o custo da latência de memória, desperdiçando o potencial dos processadores em muitos ciclos ociosos.

### 3.1.2 Outras questões importantes

Além das questões mais importantes a nível de arquitetura, existem outros pontos que devem ser considerados no projeto de um multiprocessador, para que se obtenha um ganho em desempenho proporcional aos recursos disponíveis:

- *O suporte a paralelismo implícito ou explícito*, isto é, se o usuário irá manipular ou não o paralelismo diretamente na linguagem de programação. Três possibilidades têm sido consideradas para a manipulação do paralelismo [3]:

  (a) O usuário utiliza uma linguagem seqüencial e o compilador se encarrega de detectar o paralelismo. Esta técnica é mais usada na programação de processadores vetoriais.

  (b) O usuário identifica no programa as tarefas que serão executadas em paralelo, utilizando alguma linguagem própria para processamento paralelo, mas derivada das linguagens seqüenciais. A programação de multiprocessadores convencionais é geralmente feita deste modo.

(c) O usuário utiliza uma linguagem totalmente nova, especial para processamento paralelo, na qual o paralelismo é implícito. As linguagens de fluxo de dados são incluídas nesta categoria e apesar de representarem uma modificação radical na disciplina de programação usual, conseguem expor uma quantidade maior de paralelismo. Elas são mais adequadas à programação de multiprocessadores escaláveis, porque não tornam as características da máquina visíveis ao programador [7, 9].

Uma das razões pelas quais a comunidade científica procura manter as linguagens tradicionais com todos os seus vícios e defeitos é a enorme quantidade de software existente e que não deveria ser desperdiçada. Existem propostas que se esforçam em conciliar os objetivos, projetando arquiteturas que não diferem radicalmente das atuais e que são capazes de executar os programas existentes sem muitas modificações ou perda de desempenho [18].

- *A granularidade dos processos*, ou seja, o tamanho de cada unidade de trabalho que é executada por um processador de uma só vez, sem parar para sincronização [52]. Dado o objetivo de minimizar o tempo de execução de um programa, a intenção ideal é maximizar o processamento paralelo através da diminuição do tamanho de cada unidade de trabalho, isto é, tornar o grão de paralelismo tão fino quanto possível. No entanto, isto é muito difícil em máquinas von Neumann, devido à alta sobrecarga[1] associada ao escalonamento e sincronização dos processos.

- *O método de escalonamento dos processos*, pelo qual se determina à máquina qual processo deve executar em qual processador em um determinado instante. Deve-se tomar o cuidado de escalonar a execução dos processos de modo a manter os processadores ocupados a maior quantidade de tempo possível. O escalonamento dos processos pode ser estático ou dinâmico [33]. No escalonamento estático, as tarefas são alocadas a processadores durante o projeto do algoritmo pelo programador, ou em tempo de compilação pelo compilador. Este método é o mais utilizado em arquiteturas von Neumann e apresenta a clara desvantagem do desconhecimento da história da execução do programa, o que pode levar facilmente a escolhas infelizes que deixam processadores ociosos por muito tempo. O escalonamento dinâmico, que é

[1] *overhead*

utilizado pelas arquiteturas de fluxo de dados, oferece melhor utilização dos processadores ao preço de uma sobrecarga adicional em tempo de execução.

## 3.2 Crítica às Arquiteturas de Fluxo de Dados

As arquiteturas de fluxo de dados são reconhecidas como as mais inovadoras arquiteturas paralelas para os computadores da nova geração, com desempenho esperado na ordem de teraflops. Entre as inúmeras vantagens que se atribuem ao modelo de computação de fluxo de dados pode-se destacar:

- Oferece a mais simples e mais poderosa representação de computação paralela.
- Pode explorar paralelismo na computação ao nível de arquitetura com baixa sobrecarga.
- Promete que a velocidade de computação seja limitada somente pelas dependências de dados entre as operações realizadas.

Essas vantagens são por demais teóricas e sem poder de persuasão, uma vez que são baseadas em um modelo abstrato de fluxo de dados e não em implementações reais. Há alguns anos atrás havia muito ceticismo quanto à factilidade das arquiteturas de fluxo de dados, originado do baixo desempenho apresentado em arquiteturas reais e da supremacia dos processadores vetoriais com seus compiladores vetorizadores [32].

A fim de formar uma visão criteriosa para projetar uma máquina paralela eficiente, serão examinados os principais defeitos atribuídos às atuais arquiteturas de fluxo de dados:

- Uma grande quantidade de hardware é necessária e, em particular, a memória de emparelhamento torna-se complexa se for adotado o modelo de fluxo de dados dinâmico com fichas rotuladas [70]. O emparelhamento de rótulos requer hardware especial, como memória associativa ou mecanismos de *hashing*.

  É o caso da maioria dos projetos existentes. Essa abordagem necessita de uma lógica de controle considerável e o emparelhamento consome muito tempo, tornando-se freqüentemente o gargalo do sistema.

- O tratamento destinado ao armazenamento de dados é incompleto. Como não existe a noção explícita de armazenamento de dados em posições de

memória, perdem-se todas as facilidades encontradas nos avançados sistemas de memória das arquiteturas von Neumann. Os dados são armazenados temporariamente na memória de emparelhamento, enquanto esperam seus parceiros. Este esquema torna ineficiente o manuseio de estruturas de dados e não é adequado ao processamento de listas e vetores [92]. Entre as abordagens de armazenamento de dados em fluxo de dados a que mais êxito tem obtido são as Estruturas-I [12] que, na realidade, representam um retorno ao modelo von Neumann [32].

- Um *pipeline* circular não funciona bem sob baixo paralelismo, ou seja, há uma degradação de desempenho em pontos de baixo paralelismo dentro de um programa. Isto ocorre porque as arquiteturas de fluxo de dados atuais não têm um mecanismo avançado de controle, que permita a execução determinística de certos trechos de código seqüencial. No caso de execução altamente paralela, todos os estágios de um *pipeline* circular podem ser preenchidos com fichas. Em outros casos, pode-se chegar até o extremo de somente uma ficha estar circulando no *pipeline*, conduzindo a perdas em desempenho proporcionais ao número de estágios do *pipeline*.

- Grafos de fluxos de dados puros são linguagens de máquinas incompletas e talvez até indesejáveis, embora sejam representações intermediárias poderosas para compiladores [61]. Eles são incompletos porque é difícil codificar seções críticas e operações imperativas essenciais para a execução eficiente de funções do sistema operacional, tais como mecanismos de gerenciamento de recursos. Eles podem ser indesejáveis porque implicam um escalonamento dinâmico uniforme para todas as instruções, impedindo um compilador de expressar um escalonamento estático capaz de resultar em maior eficiência em tempo de execução, reduzindo a sincronização redundante de operandos e permitindo a utilização de registradores de alta velocidade para comunicação entre instruções.

- O número de instruções executadas é consideravelmente maior que em computadores von Neumann. Avaliações usando situações reais revelaram que, para um mesmo programa escrito em Fortran, uma máquina de fluxo de dados executa cerca de duas vezes mais instruções do que uma máquina von Neumann [11]. O controle distribuído, característica marcante do modelo de fluxo de dados, implica que cada dado deve ser controlado individualmente, atividade que consome muitas instruções consideradas sobrecarga pura. Além disso, a ausência de armazenamento explícito tem enorme impacto sobre o número de instruções executadas. Uma vez que os valores

são passados de uma instrução a outra, valores que não mudam durante a computação de uma iteração para outra devem circular obrigatoriamente no sistema. Muitas também são as instruções necessárias somente para fazer cópias de dados quando um determinado valor é requerido por várias instruções.

- O paralelismo de granularidade fina causa aumento de sobrecarga no processamento paralelo. Aquilo que sempre foi considerado uma das principais vantagens do modelo de fluxo de dados, representa, na verdade, um enorme desperdício de tempo e recursos. Sendo todas as instruções consideradas processos distintos que se comunicam através de passagem de mensagens, existe um gasto de escalonamento, sincronização, execução e comunicação para cada instrução executada. Infelizmente, é impossível escapar disto, mesmo quando um bom escalonamento estático é conhecido em tempo de compilação.

- O tráfego de pacotes é muito pesado no sistema e freqüentemente a rede de comunicação torna-se um sério gargalo. Mesmo que o anel e a rede de interconexão possam operar a velocidades muito altas, eles estarão sempre limitados pela tecnologia dos dispositivos. Esta pode ser a causa direta do problema anterior e somente pode ser corrigida aumentando-se a granularidade do paralelismo, contribuindo assim para a diminuição do número de pacotes a serem transmitidos.

## 3.3 A união dos modelos

Historicamente, os modelos de computação von Neumann e de fluxo de dados têm sido vistos como radicalmente diferentes, e talvez até irreconciliáveis. Por vários anos, pesquisadores tentaram descobrir qual dos modelos representa uma base melhor para os futuros sistemas de computação paralelos de alto desempenho. Atualmente está se tornando cada vez mais claro que os dois modelos não são ortogonais, mas representam os extremos opostos de um grande espectro de arquiteturas [61, 48, 34, 11, 18, 39]. Isto leva a crer que existe um ponto ótimo entre esses dois extremos, ou seja, uma arquitetura híbrida que combine sinergeticamente as melhores características dos modelos von Neumann e de fluxo de dados. Esta pode ser vista como uma evolução de fluxo de dados em direção a um melhor controle sobre a execução das instruções e aproveitamento mais eficiente dos recursos de hardware, ou como uma evolução de von Neumann em direção a um melhor suporte de hardware para sincronização e tolerância a operações de

memória de longa latência. Como um resultado, o tamanho do grão de paralelismo que uma arquitetura híbrida deste tipo pode suportar é flexível, e técnicas de compilação e de hardware podem ser combinadas para descobrir o tamanho efetivo do grão que é necessário para a exploração eficiente do paralelismo. Uma arquitetura que seja capaz de explorar eficientemente todo o paralelismo disponível em um problema vem sendo um tema de pesquisa no mundo inteiro há vários anos.

As arquiteturas von Neumann não apresentam desempenho satisfatório quando dispostas em uma configuração multiprocessadora por não proverem suporte eficiente para as questões de latência e sincronização. No entanto, é importante observar que esses argumentos favorecem a alteração do mecanismo básico von Neumann, e não o seu completo abandono. Em situações onde a seqüência de instruções e as restrições quanto às dependências de dados podem ser identificadas em tempo de compilação, ainda existem razões para acreditar que o estilo seqüencial proporciona melhor controle sobre o comportamento da máquina do que os mecanismos de escalonamento dinâmico, e, por este motivo, melhor relação custo/desempenho. Não se pode ignorar a simplicidade e eficiência do mecanismo de seqüenciamento de instruções do modelo von Neumann. Nem tampouco, pode-se desprezar as lições aprendidas em mais de 40 anos de otimizações na arquitetura para um melhor aproveitamento dos recursos de hardware. Deve-se considerar que, apesar do desejo de revolucionar a área de arquitetura de computadores, a base mais bem entendida para a sua construção ainda são as máquinas von Neumann.

Apesar de o modelo abstrato de fluxo de dados sugerir uma representação mais poderosa para o processamento paralelo, implementações reais de máquinas de máquinas de fluxo de dados não foram capazes de transformar os anseios dos pesquisadores em realidade. Isto deve-se em parte ao pouco tempo de pesquisa na área, pois muito tem-se ainda a aprender. Mas, a principal causa foi o desejo de uma separação radical do modelo von Neumann, com o argumento de que o estilo de execução seqüencial é a antítese do paralelismo. Esta é a causa da maioria das deficiências das arquiteturas de fluxo de dados que foram apresentadas.

Em um programa de fluxo de dados, o tempo para executar as instruções do caminho crítico é o produto do tamanho do caminho crítico pelo comprimento do *pipeline*. Isto pode ser otimizado, realizando-se a sincronização explicitamente onde for necessária e confiando-se em mecanismos mais tradicionais e mais bem entendidos para o seqüenciamento das instruções nos outros casos. Além disso, o comprimento do *pipeline* pode ser diminuído se forem identificados estágios desnecessários, e o número de instruções pode ser reduzido se os dados forem armazenados em memória, não sendo necessárias instruções para copiá-los expli-

citamente.

Dois pontos têm se revelado de importância vital para as arquiteturas híbridas: o modelo de armazenamento de dados e o mecanismo de escalonamento e seqüenciamento de instruções, que são discutidos a seguir.

### 3.3.1 O modelo de armazenamento

Nas arquiteturas que seguem o modelo de fluxo de dados dinâmico baseado em fichas rotuladas, o gerenciamento de memória é efetuado implicitamente pelo hardware da máquina. Uma falha em encontrar o parceiro na memória de agrupamento aloca espaço para um dado automaticamente. A simples criação de um novo rótulo implica na existência de recursos de memória para o armazenamento dos dados de uma determinada ativação de um bloco de código.

As implementações seqüenciais baseadas no modelo von Neumann deixam o gerenciamento de memória a cargo do compilador, utilizando pilhas de registros de ativação para conter as variáveis de um bloco de código invocado. Como há um único foco de controle, isto é, somente o bloco de código invocado está ativo, o registro de ativação em questão sempre é aquele que está no topo da pilha.

Em ambientes de fluxo de dados é imprescindível que haja múltiplas invocações de blocos de código ativos concorrentemente. O modelo de pilha não é apropriado para este fim, pela restrição da disponibilidades única e exclusiva do registro do topo da pilha. Por outro lado, o modelo de gerenciamento implícito também já demonstrou deficiências, por desperdiçar muitos recursos de hardware. A abordagem que tem sido utilizada na maioria das implementações de arquiteturas híbridas consiste em generalizar o modelo tradicional de pilha para uma árvore de registros de ativação. Ao compilador é atribuída a tarefa de gerenciamento de memória a fim de simplificar o hardware e o sistema de tempo de execução. Desse modo, quando uma função é invocada, um registro de ativação é alocado implicitamente para ela. Na realidade, este serviço é feito por instruções especiais inseridas no código pelo compilador, que são interpretadas pelo sistema de memória em tempo de execução. Cada instrução especifica a posição dos seus operandos dentro do registro de ativação. O endereçamento dos dados é relativo à base do registro de ativação, seguindo o modelo *base + deslocamento* utilizado para o gerenciamento de memória virtual nos sistemas operacionais. Em tempo de execução, a invocação assíncrona concorrente de vários blocos de código representa uma figura que pode ser vista como uma árvore formada pelos registros de ativação.

### 3.3.2 Arquiteturas *multithreaded*

Para reduzir o custo da latência de memória, é essencial que o processador seja capaz de executar múltiplos acessos à memória sobrepostos e receber as respostas em qualquer ordem. Arquiteturas von Neumann não possuem esta capacidade, porque são limitadas pelas restrições impostas pelo seqüenciamento das instruções. Uma maneira de solucionar esta deficiência é entrelaçar a execução de muitos *threads* seqüenciais e proporcionar um mecanismo de sincronização de baixo nível eficiente para receber as respostas da memória possivelmente fora de ordem. Máquinas que suportam muitos *threads* se parecem-se cada vez menos com as máquinas von Neumann, à medida que o número de *threads* aumenta, e são chamadas *multithreaded.* O Denelcor HEP [77] pode ser considerado o primeiro computador *multithreaded* disponível comercialmente.

Arquiteturas de fluxo de dados podem ser consideradas como um exemplo extremo de máquinas com múltiplos *threads*, nas quais cada instrução constitui um *thread* independente e somente *threads* não suspensos são escalonados para a execução. No entanto, como já foi mencionado, grafos de fluxo de dados são linguagens de máquina incompletas e até indesejáveis. Isto conduz à necessidade de modificar a visão de como uma máquina de fluxo de dados funciona. Em vez de considerar o progresso computacional como o consumo de fichas e o disparo de instruções habilitadas, raciocina-se acerca da evolução de múltiplos *threads* interativos, onde as operações de *fork* e *join* são extremamente eficientes.

Arquiteturas *multithreaded* constituem o tema de vários projetos de pesquisa, tanto derivados de arquiteturas von Neumann [77, 2, 4, 44, 45], quanto de arquiteturas de fluxo de dados [72, 57, 58, 81, 61, 46]. Ambas as abordagens constituem modelos de execução híbridos, a diferença entre elas é uma questão de ênfase. Os modelos *multithreaded* von Neumann empregam novas técnicas para reduzir o custo da sincronização de granularidade fina, enquanto mantêm a eficiência do escalonamento seqüencial de instruções. Os modelos *multithreaded* de fluxo de dados tentam melhorar a eficiência da computação, enquanto mantêm a capacidade de realizar sincronização de maneira extremamente rápida.

Papadopoulos e Culler [61] argumentam que *multithreading* é um reconhecimento do fato empírico de que paralelismo a nível de instrução é provavelmente melhor explorado no contexto de um *thread* executando em um *pipeline* bem projetado, do que na execução dinâmica do grafo de fluxo de dados equivalente. Alguns resultados mostram que em muitos casos o estilo *multithreaded* resulta em menos ciclos de máquina para executar um dado programa, se comparado ao estilo tradicional de fluxo de dados [81]. A razão para este sucesso é que ao separar o fluxo dos dados do fluxo de controle podem ser eliminados fluxos de

controle redundantes. Em outras palavras, menos operações de *fork* (criação de fichas) e *join* (emparelhamento de fichas) são necessárias.

Culler [24, 73] argumenta que isentar o compilador da responsabilidade de escalonar entidades de programa de baixo nível esbanja recursos críticos do processador e impõe cargas excessivas ao hardware. Mantendo parte desse controle, o processador pode otimizar o uso dos seus recursos para o caso esperado, e não para o pior caso. Então, tolerância para latência e sincronização barata requerem que o compilador adote uma representação de programa conveniente, mas isto não precisa estar contido na arquitetura do processador. Para pesquisar esta idéia, foi formulada uma máquina abstrata primitiva, na qual os recursos do processador e o escalonamento de *threads* são explícitos ao nível de instrução, e todo gerenciamento de memória é de responsabilidade da linguagem, e não da máquina. O objetivo desta máquina abstrata é identificar qual suporte de arquitetura é mais importante para executar programas grandes em máquinas paralelas.

## 3.4 Propostas

Existem atualmente várias propostas que procuram reunir em uma arquitetura híbrida as melhores características dos modelos von Neumann e fluxo de dados. Nem todas utilizam o termo "híbrida" para caracterizar sua arquitetura e algumas apresentam maior tendência a um dos modelos, incorporando apenas timidamente características importantes do outro. A seguir são apresentadas 4 arquiteturas que possuem como intenção de projeto a integração dos modelos. DFVN e P-RISC partem do modelo von Neumann para torná-lo mais adequado ao processamento paralelo, enquanto que MDFA e Monsoon tomam como base o modelo de fluxo de dados, com a intenção de aumentar sua eficiência.

### 3.4.1 DFVN

A arquitetura híbrida *fluxo de dados/von Neumann* [48] (Data Flow/Von Neumann — DFVN) parte da premissa de que o modelo básico von Neumann deve ser alterado e não completamente abandonado.

Para atingir este objetivo foram desenvolvidos uma arquitetura de processador e uma linguagem de máquina apropriadas. A linguagem de máquina paralela pode ser vista como um conjunto de instruções que incorpora as noções de tempo limitado para a execução das instruções (ou seja, independente de latência), um grande espaço de nomes para sincronização e permite expressar os conceitos de sincronização implícita e explícita. Dessa forma, é possível indicar grupos de instruções seqüenciais que formam uma unidade escalonável. Em DFVN esses

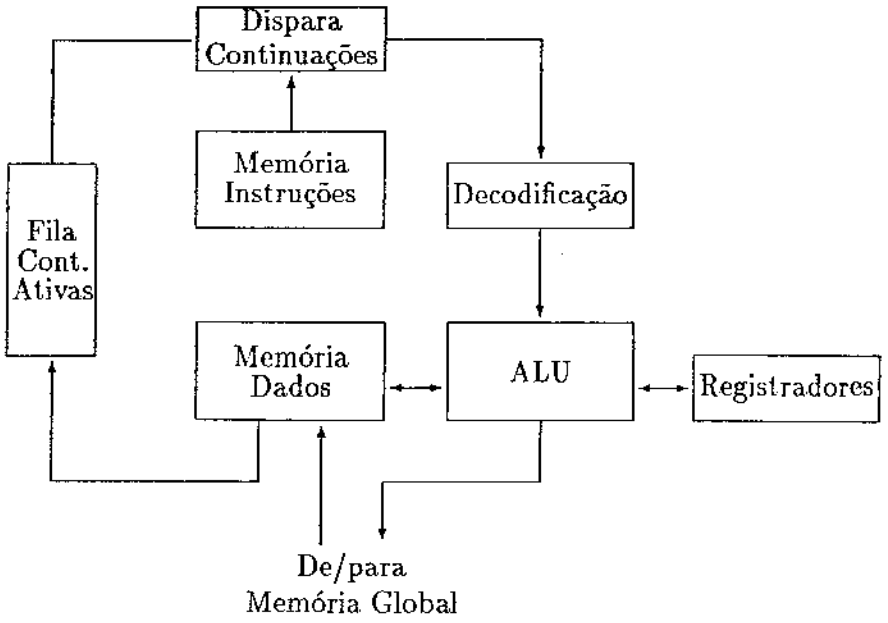



Figura 3.1: O processador híbrido DFVN

grupos de instruções são chamados de SQ (scheduling quanta). Os SQs são gerados pelo compilador, de acordo com o critério de otimização escolhido, e seu escalonamento é executado com base na disponibilidade dos dados.

Cada processador da arquitetura híbrida (Figura 3.1) é composto de um *pipeline* de dados, uma coleção de registradores e uma memória local. O hardware é bastante similar a uma arquitetura von Neumann convencional, mas com algumas diferenças importantes. É introduzido um novo tipo de dado, chamado continuação, que é uma tupla formada de um contador de programa (PC) e um registrador de base do registro de ativação. Continuações podem assumir os estados lógicos *habilitada*, *executando* e *suspensa*.

Instruções podem referir-se a operandos em registradores ou em posições da memória local de dados, que possui bits de presença e um comportamento similar às Estruturas-I. Referências a posições da memória podem ser não-sincronizantes, que se comportam como operações normais de leitura, ou sincronizantes, que implicam na suspensão da continuação que está executando se a posição estiver vazia. Quando uma continuação é suspensa, ela é armazenada na posição de memória que gerou a suspensão e outra é retirada da fila de continuações ativas. Após a condição de suspensão ser satisfeita, o que ocorre quando uma outra

continuação escreve na posição, a continuação é retirada da posição de memória e colocada na fila.

Uma continuação é invocada quando um SQ é invocado e pode ser suspensa várias vezes até todas as instruções do SQ serem executadas. Enquanto o SQ não terminar ou a continuação não for suspensa, o PC da continuação que está executando é incrementado à maneira von Neumann.

#### Comentários

DFVN pode ser considerada um dos marcos iniciais na área de pesquisas em arquiteturas híbridas, porque mostrou intenção explícita de unir as melhores características dos modelos fluxo de dados e von Neumann. No entanto, na ânsia de encontrar a arquitetura ideal, criou-se uma arquitetura bastante complexa para suportar o modelo de execução versátil projetado. Este é o caso da possibilidade de haver sincronização em cada instrução de um SQ, desde que a posição endereçada na memória esteja vazia. Não se pode garantir que um SQ irá executar até o fim de uma só vez e embora cada SQ possua em PC este não indica necessariamente um escalonamento von Neumann. A suspensão e retomada de execução muito freqüentes de um SQ podem levar a uma perda considerável em desempenho, pois cada suspensão gera uma bolha no *pipeline* e valores de registradores não são preservados após uma suspensão. Além disso, o número de instruções executadas em programas reais não difere muito daquele constatado em TTDA [11], de modo que neste sentido não há uma aproximação em relação às máquinas von Neumann. DFVN não chegou a ser construída, e talvez nem fosse viável. Mas o extenso estudo que foi realizado para o seu projeto tem influenciado a maioria das propostas de arquiteturas híbridas atuais.

### 3.4.2 P-RISC

P-RISC [57](de Parallel RISC) é uma arquitetura com capacidade de execução baseada em fluxo de dados de granularidade fina, mas é em princípio uma arquitetura RISC *multithreaded*. Ela pode explorar tecnologia de compilação convencional e de fluxo de dados e, mais do que isto, pode ser vista como uma máquina de fluxo de dados que consegue obter compatibilidade de software com máquinas convencionais von Neumann.

A arquitetura de P-RISC é composta de uma coleção de EPs (Elementos de Processamento) interconectados a Elementos de Memória Heap com comportamento similar a Estruturas-I. Cada EP (Figura 3.2) tem memória local para código e registros de ativação, e executa *threads* descritos por um ponteiro de

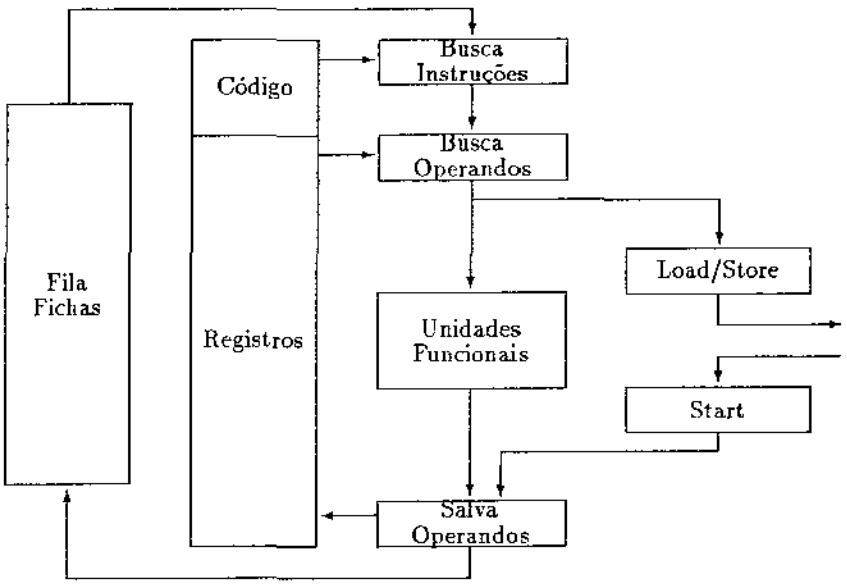



Figura 3.2: Elemento de processamento de P-RISC

instrução (IP) e um ponteiro de registro de ativação (FP). Esse par é chamado de ficha ou continuação e reside na fila de fichas. Para executar uma instrução, o processador retira uma ficha $< FP.IP >$ da fila, a executa e gera outra ficha contendo $< FP.IP + 1 >$, ou seja, o escalonamento das instruções dentro de um *thread* é implícito, à maneira von Neumann.

Para que o processador possa criar e sincronizar *threads* e assim possuir várias referências pendentes à memória, são incorporadas duas instruções: *fork* e *join*. *Fork end* produz duas fichas, na forma $< FP.IP+1 >$ e $< FP.IP+end >$. *Join end* produz uma ficha $< FP.IP+1 >$ quando o conteúdo de *end* é 1, e não produz nenhuma ficha quando o conteúdo de *end* é 0. Nesse caso o *thread* é destruído e o conteúdo de *end* muda para 1. Desse modo, P-RISC tem a capacidade de executar grafos de fluxo de dados, embora não possua memória de sincronização.

P-RISC utiliza instruções de 3 endereços, 2 para leitura de operandos e 1 para a escrita do resultado, de maneira idêntica às arquiteturas RISC convencionais. Os acessos à memória de dados são executados em estágios específicos do *pipeline*, de modo que o efeito da latência é escondido das unidades funcionais. A execução dos *threads* é entrelaçada com base em instruções, mas existe a possibilidade de um aprimoramento em direção à execução contínua das instruções de um *thread*

para melhor explorar a utilização de registradores de alta velocidade.

**Comentários**

P-RISC não tem a intenção de ser um projeto completo, com possibilidade de implementação. É apenas uma proposta preliminar que necessita de mais detalhamento para poder ser avaliada com maior clareza, mas trouxe à discussão várias questões importantes para servirem de estímulo a pesquisas nesta direção. A maior contribuição de P-RISC é a idéia de que se pode obter todos os benefícios do modelo de fluxo de dados sem um conjunto de instruções poderoso e uma arquitetura complexa. Um reflexo dessa abordagem pode ser sentido na inexistência de bits de presença para sincronização na memória de registros de ativação. Em vez disso, algumas posições da memória são interpretadas como estados vazio/cheio para sincronização, que ocorre somente em instruções *join*. Então, P-RISC é uma arquitetura mais simples, mas executa mais instruções, uma vez que a sincronização é separada das instruções aritméticas.

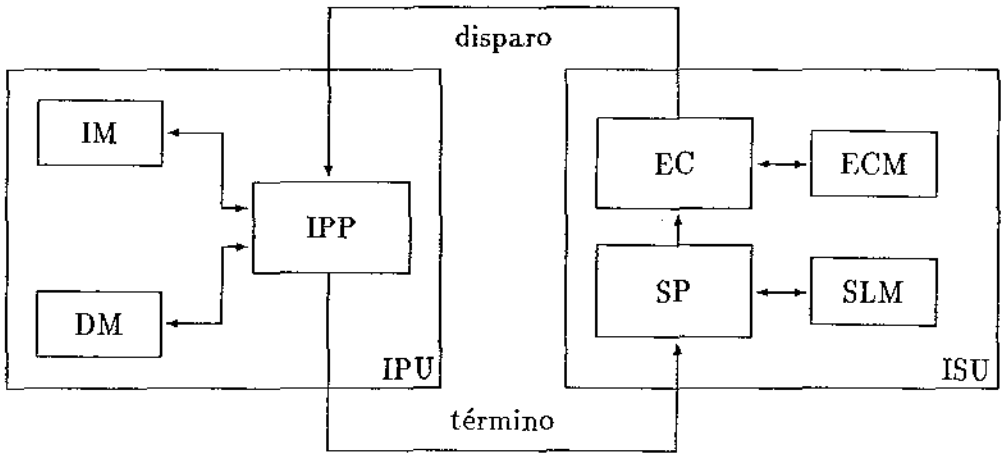
### 3.4.3 MDFA

MDFA (McGill Dataflow Architecture)[34] é uma arquitetura baseada no princípio *argument-fetching*, onde o escalonamento das instruções é separado do caminho de execução de dados, produzindo um modelo de fluxo de dados que permite facilmente a extensão para uma arquitetura híbrida. A partir do modelo MDFA estão sendo projetados variantes híbridos para utilizá-los como veículo de estudo a uma série de questões sobre arquiteturas *multithreaded*, assim como técnicas de compilação para elas.

Um programa em MDFA é uma tupla $P, S$ onde a porção $P$, chamada código-$P$, é um conjunto de instruções de 3 endereços. A porção $S$, chamada código-$S$, é um grafo direcionado de sinalização.

A arquitetura (Figura 3.3) consiste de uma Unidade de Processamento de Instruções (IPU) e uma Unidade de Escalonamento de Instruções (ISU). As instruções, no código-$P$, chamadas instruções-$p$, são similares àquelas de uma arquitetura convencional e são armazenadas na IM. Os operandos e resultados de instruções são armazenados na DM.

Quando um sinal de disparo $< i, b >$ (onde $i$ é o endereço da instrução-$p$ e $b$ é o endereço base do quadro de ativação) está presente na entrada da IPU, a instrução-$p$ correspondente é executada em uma maneira similar à von Neumann, passando através dos estágios normais de busca de instrução, busca de operandos, execução e armazenamento de resultado, além de ter acesso às




IPU: unidade de processamento de instruções
IPP: pipeline de processamento de instruções
IM: memória de instruções
DM: memória de dados
ISU: unidade de escalonamento de instruções
SP: processador de sinais
SLM: memória de lista de sinais
EC: controlador de habilitação
ECM: memória de contagem de habilitação

Figura 3.3: A arquitetura da MDFA

memórias apropriadas. A maior diferença é que após o término da execução, um sinal de término $< i, b >$ é liberado pela CPU.

O escalonamento da execução das instruções-$p$ é realizado na ISU pelo processamento do código-$S$, que consiste de um conjunto de instruções-$s$, interconectadas por arestas direcionadas. Cada instrução-$s$ contém uma lista de endereços de instruções-$s$ que são destinos das suas arestas de saída. Esta lista está armazenada na SLM. Além disso, para cada instrução-$s$ existe um contador de escalonamento, que indica quantos sinais uma instrução-$s$ deve ainda receber para ser habilitada e está armazenado na ECM. A habilitação de uma instrução-$s$ implica na geração de um sinal de disparo $< i, b >$ para executar a instrução-$p$ correspondente.

Para tornar MDFA uma arquitetura híbrida é necessário prover uma alternativa para a continuação das instruções, isto é, uma maneira de a IPU gerar diretamente o endereço da próxima instrução-$p$, em vez de tomar um endereço escalonado pela ISU. Para implementar este modelo híbrido, cada instrução-$p$ é estendida com um campo extra, chamado bit von Neumann, que indica se a instrução está seguindo o estilo de escalonamento von Neumann ou de fluxo de

dados. Enquanto o bit indicar estilo von Neumann as instruções são executadas seqüencialmente; quando o bit indicar estilo fluxo de dados, o sinal de término é gerado e uma nova instrução escalonada pela ISU é executada.

**Comentários**

MDFA apresenta uma característica inovadora: a eliminação das bolhas no *pipeline* devido ao agrupamento de operandos. O mecanismo de escalonamento de instruções de fluxo de dados é claramente separado do processamento das instruções. Então, MDFA não realiza o agrupamento no caminho crítico de execução, seja explícita ou implicitamente. Bolhas nunca irão ocorrer se a ISU puder tratar os sinais gerados pela IPU rápido o suficiente para esconder o custo do agrupamento e não deixar faltar serviço para a IPU.

A proposta de separar o escalonamento e a execução das instruções em duas unidades distintas (ISU e IPU) consegue aparentemente obter um melhor rendimento do modelo híbrido de execução. A ISU é a unidade de fluxo de dados, que apenas decide qual a próxima instrução, ou grupo de instruções, a executar e envia um sinal à IPU. A IPU é a unidade von Neumann, que executa grupos de instruções seqüencialmente, com base no escalonamento efetuado pela ISU.

Resultados de simulação demonstraram que havendo suficiente paralelismo na aplicação a ISU consegue um bom escalonamento, nunca deixando a IPU ociosa. Mas, além disso é necessário que a execução na IPU seja eficiente, levando-se em conta que cada instrução realiza 3 acessos à memória. Esta questão está sendo tratada em SAM [46] (Super-Actor Machine), uma arquitetura *multithreaded* que utiliza um mecanismo chamado Registrador-Cache para reduzir o tempo de latência da memória.

### 3.4.4 Monsoon

Monsoon [60, 59] é um multiprocessador de propósito geral que incorpora um armazenamento explícito de fichas (ETS [25] — Explicit Token Store). A idéia central no modelo ETS é que o armazenamento para fichas é dinamicamente alocado em blocos, com a utilização detalhada das posições dentro dos blocos determinada em tempo de compilação. Monsoon foi quem primeiro introduziu o modelo de armazenamento explícito de fichas baseado em registros de ativação, que está sendo utilizado atualmente na maioria das arquiteturas híbridas.

Uma ficha em Monsoon é composta de um valor, um apontador para a instrução a executar (IP) e um apontador para um registro de ativação (FP). Os dois últimos formam o rótulo. A instrução obtida a partir da posição IP especifica um código de operação, um deslocamento no registro de ativação para o

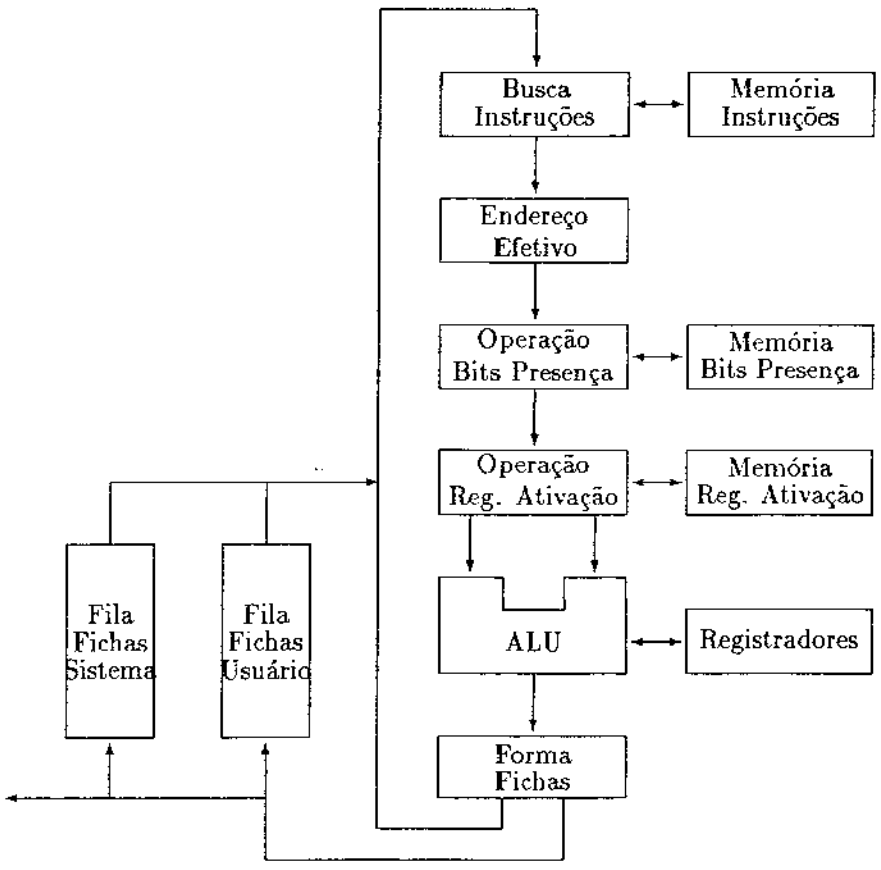



Figura 3.4: Elemento de processamento de Monsoon

emparelhamento e uma ou duas instruções de destino. Cada posição dentro do registro de ativação possui bits de presença, sendo que com uma simples transição de estados nestes bits é possível executar as sincronizações que tornam possível seguir a regra de disparo de fluxo de dados dinâmico.

Um Elemento de Processamento (Figura 3.4) de Monsoon é um processador com um *pipeline* de oito estágios. A operação do *pipeline* inicia-se com a busca da instrução especificada pelo IP da ficha e o cálculo do endereço efetivo da posição no registro de ativação. Em seguida os bits de presença da posição são lidos, modificados e escritos de volta. Se eles indicarem *vazio* a operação é interrompida, porque este é o primeiro operando de uma instrução que necessita de dois operandos. Neste caso, uma bolha é produzida e transferida aos estágios restantes do *pipeline*, não havendo chance para a recuperação. Se os bits de presença

indicarem outros estados possíveis, a parte do valor da posição do registro de ativação especificada é ignorada, lida, escrita ou trocada com o valor da ficha. Os estágios restantes representam operações da ALU e de criação de fichas.

A cada ciclo do processador uma ficha entra no topo do *pipeline* e, ao final de oito ciclos, emergem do fundo zero, uma ou duas fichas, que são armazenadas nas filas do usuário ou do sistema, dependendo do caso, ou imediatamente recirculadas ao topo do *pipeline*. Esta característica permite a execução seqüencial de código seguindo o modelo de escalonamento von Neumann. *Threads* podem ser gerados pelo compilador e as fichas são sempre circuladas diretamente sem entrar em filas, ignorando inclusive os bits de presença do quadro de ativação. Monsoon permite a execução entrelaçada de 8 linhas de instruções simultaneamente e cada *thread* pode dispor de três registradores para a comunicação de dados rápida entre as instruções. Quando um *thread* acaba, suas fichas são armazenadas nas filas e uma nova ficha é retirada. Neste sentido, embora seja uma arquitetura de fluxo de dados, Monsoon pode ser considerada uma máquina *multithreaded* com capacidade de gerenciar um número quase infinito de linhas de execução, limitado apenas pela capacidade das filas.

### Comentários

Apesar de possuir características híbridas, Monsoon é basicamente uma arquitetura de fluxo de dados, no sentido de que fichas não somente escalonam instruções, mas também carregam dados. Ela é uma máquina de um endereço, o que é suficiente para a execução em fluxo de dados, mas impossibilita um gerenciamento eficiente de memória necessário para execução à von Neumann. Uma operação de *store* não pode ser feita explicitamente; para isto é necessária a execução de uma instrução que deposite o dado na posição do registro de ativação e seja então abortada, criando uma bolha. Numa avaliação comparativa com o Cray-1, Monsoon executou pelo menos duas vezes mais instruções.

Em Monsoon oito *threads* devem estar executando simultaneamente para manter o *pipeline* cheio e obter o desempenho integral do processador. Isto torna-se uma preocupação, uma vez que uma configuração com $n$ processadores necessita de um paralelismo de $8n$ para que não haja queda do desempenho. Embora este entrelaçamento de granularidade fina de múltiplos *threads* evite os *hazards* [26] no *pipeline*, isto contribui para um baixo desempenho no caso de um único *thread* [2].

Foi desenvolvido também um modelo *multithreaded* para Monsoon, sem, no entanto, recorrer a muitas modificações no conjunto de instruções e na arquitetura [61]. Monsoon, ao nível de macroarquitetura passa a ser considerada a

máquina *multithreaded* que mais facilmente harmoniza os modelos von Neumann e de fluxo de dados. Os resultados obtidos com essa experiência levaram à criação de uma nova máquina, chamada *T [58], uma arquitetura *multithreaded* dedicada à computação maciçamente paralela. Ela é considerada o último passo evolutivo iniciado com TTDA, passando por Monsoon e com influências de DFVN e P-RISC.

### 3.4.5 Outras propostas

O trabalho de Buehrer e Ekanadham [18] foi um passo importante na exploração do uso de estruturas de fluxo de dados em uma arquitetura von Neumann. APRIL [2] é uma arquitetura paralela *multithreaded* que procura otimizar o desempenho de *threads* individuais sem o entrelaçamento fino com base em instruções. TERA [4] é uma arquitetura *multithreaded* superescalar com suporte para entrelaçamento fino de instruções e facilidades para a implementação de linguagens não-numéricas e aplicativas. MASA [44] tem como principal objetivo a execução de programas escritos em LISP através de uma arquitetura *multithreaded.*

EM-4 [70, 89, 72] é uma arquitetura de fluxo de dados que introduz o modelo de aresta fortemente conexa para executar uma seqüência de instruções com apenas uma sincronização utilizando registradores para comunicar resultados entre elas. Também emprega agrupamento direto, mas de forma restritiva, pois as instruções não podem referir-se diretamente a posições dentro do registro de ativação.

Epsilon-2 [39] é uma arquitetura híbrida — evolução da máquina de fluxo de dados estática Epsilon [38] e versões anteriores de ETS — que possui suporte para execução de grãos de instruções de tamanho variado. Um mecanismo de "repetição na entrada" permite uma seqüência de instruções ser iniciada para execução contínua. Epsilon-2 possui memória de sincronização separada do armazenamento de dados, permitindo maior flexibilidade na execução.

Harray [90] e Cydra 5 [64] unem características de von Neumann e fluxo de dados. No entanto, não podem ser consideradas arquiteturas híbridas, de acordo com os critérios utilizados neste trabalho, pois são muito restritivas e voltadas à execução de programas em FORTRAN.

## 3.5 Resumo

Este capítulo mostrou como a união das melhores características das arquiteturas von Neumann e de fluxo de dados pode gerar uma arquitetura híbrida capaz de apresentar um desempenho muito superior aos atuais supercomputadores. Para

isto, primeiramente foram analisadas as deficiências dos dois modelos. Processadores von Neumann, quando ligados em uma configuração paralela, sofrem grandes penalizações devido a duas questões fundamentais: latência de memória e sincronização. Por outro lado, as arquiteturas de fluxo de dados apresentam soluções elegantes para estas questões, mas esbarram em outros problemas que já foram resolvidos há muito tempo pelas arquiteturas von Neumann.

Uma arquitetura híbrida pode ser vista como uma evolução de von Neumann em direção a um melhor suporte de hardware para sincronização e tolerância a operações de memória de longa latência, ou como uma evolução de fluxo de dados em direção a um melhor controle sobre a execução das instruções e aproveitamento mais eficiente dos recursos de hardware Um dos pontos fundamentais é o modelo de armazenamento utilizado, pois nem o modelo de pilha von Neumann, nem o modelo de alocação implícita de fluxo de dados, são adequados. Em geral, arquiteturas híbridas utilizam um modelo de armazenamento baseado em árvores de registros de ativação. Além disso, é necessário um mecanismo de execução de linhas seqüenciais de controle, que ofereça uma solução eficiente aos problemas de ambos os modelos estudados. A utilização de tal mecanismo caracteriza as arquiteturas chamadas *multithreaded.*

Ao final do capítulo, foram apresentadas quatro arquiteturas cujo projeto baseia-se na integração dos modelos. DFVN e P-RISC partem do modelo von Neumann para torná-lo mais adequado ao processamento paralelo, enquanto Monsoon e MDFA tomam como base o modelo de fluxo de dados, com a intenção de aumentar a sua eficiência.

# Capítulo 4

# A arquitetura MX

Neste capítulo serão discutidas questões relacionadas à inclusão de uma memória de armazenamento de dados em uma máquina de fluxo de dados. De acordo com as idéias apresentadas no Capítulo 3, pode-se ver que a implementação do modelo de fluxo de dados dinâmico baseado em fichas rotuladas apresenta um fraco desempenho, devido principalmente ao hardware complexo utilizado no agrupamento dos dados. Acredita-se que eliminando a memória associativa do agrupamento e tornando a alocação e ocupação da memória explícitas a cargo do compilador, muitos benefícios podem ser alcançados. O ideal pretendido é a eliminação das deficiências tradicionais das arquiteturas de fluxo de dados e a incorporação das melhores características do modelo von Neumann, como a utilização otimizada da memória e a execução de blocos seqüenciais de código.

O capítulo inicia com a avaliação da máquina de Manchester, realçando seus pontos positivos e negativos, para que com base nas lições aprendidas neste projeto seja apresentada em seguida uma nova arquitetura —aqui denominada MX— capaz de executar o modelo de fluxo de dados dinâmicos e que segue as tendências atuais de pesquisa nesta área. Por fim, são feitas considerações a respeito das alterações de software necessárias para a programação desta máquina.

## 4.1 Avaliação da MFDM

A Máquina de Manchester foi construída para demonstrar a factibilidade de uma arquitetura dirigida pelos dados e não para obter um alto desempenho [50]. Mesmo assim, o desempenho observado, entre 1 e 2 Mips, não é considerado muito animador. Isto foi em parte devido à tecnologia TTL utilizada no protótipo, cuja substituição pela tecnologia ECL mais veloz disponível naquela época poderia

levar a um desempenho aproximado de 20 Mips [86]. No entanto, somente utilizar uma tecnologia mais veloz não é a solução. Em especial deseja-se extrair o máximo desempenho possível de uma dada tecnologia. Em Manchester foi feita a opção por componentes lentos e baratos para que os gargalos do sistema fossem descobertos e retirados. Muito valiosas foram as contribuições oferecidas pelo grupo que projetou e construiu a Máquina de Manchester e com base nos resultados do seu trabalho pode-se agora avaliar o projeto, identificando suas boas e más características.

### 4.1.1 Pontos negativos

Uma série de deficiências podem ser observadas na MFDM, quando se tem a intenção de tomá-la como base para o projeto de um sistema de alto desempenho. Algumas destas deficiências são comuns a todas as arquiteturas que implementam o modelo de fluxo de dados dinâmico, e já foram examinadas genericamente no Capítulo 3. Aqui são vistas quais as suas conseqüências específicas na arquitetura da MFDM.

- *O tamanho da ficha é limitado em 96 bits.* Isto permite apenas 32 bits no campo destinado aos dados, o que torna-se um problema particularmente quando se requer maior precisão para aplicações numéricas. Simplesmente aumentar o tamanho da ficha não é uma solução razoável para este problema, pois os tamanhos dos barramentos entre todas as unidades devem também ser aumentados, e isto não é sempre possível devido a restrições tecnológicas ou de padronização. São necessárias modificações na arquitetura para que qualquer tipo de dado possa ser tratado na medida necessária sem grandes perdas em desempenho nem alterações no tamanho dos barramentos.

- *O tratamento de estruturas de dados é inadequado.* No projeto original, estruturas de dados eram armazenadas na Unidade de Agrupamento utilizando funções especiais [20]. Isto apresenta uma vantagem, pois desaparece o problema da alocação dinâmica de espaço. No entanto, contribui bastante para o congestionamento da unidade e causa degradação considerável em desempenho. A Unidade de Armazenamento de Estruturas foi criada justamente para solucionar este problema, mantendo as estruturas de dados em uma memória particular fora do anel. Mas mesmo esta solução parece não ser o ideal, pois o tratamento a dados estruturados é totalmente diferente daquele destinado a dados escalares, que circulam no anel como fichas. A realização de qualquer operação sobre uma estrutura de dados requer que

todos os seus elementos sejam transformados em dados escalares e circulados no anel, para depois serem novamente armazenados. Esta abordagem apesar de ser a mais aceita atualmente, introduz uma sobrecarga muito grande na manipulação de estruturas de dados.

- *O agrupamento de dados é baseado em memória pseudo-associativa.* A Unidade de Agrupamento é considerada o gargalo do sistema pois a taxa máxima com que consegue realizar o emparelhamento define a velocidade de pico da máquina [43]. Também é responsável pela criação das bolhas que causam uma queda no desempenho[1] do sistema à medida que mais processadores são acrescentados. A Unidade de Agrupamento tem recebido várias críticas e há algumas propostas para a sua divisão em várias subunidades menores visando melhorar seu desempenho e recuperar as perdas com as bolhas [37, 50, 62]. Existe mais um agravante que é o fato dos transbordamentos[2] serem tratados por um processador separado. Levando em conta que a partir de 25% de ocupação da Unidade de Agrupamento começam a ocorrer transbordamentos, está claro que ao mesmo tempo que o desempenho do sistema é gravemente afetado há um desperdício inaceitável de memória. Os programas executados na MFDM não foram grandes o suficiente para apresentar transbordamentos, de modo que não se tem dados concretos a respeito do prejuízo que isto acarreta. Mesmo assim, pode-se avaliar que um processador separado para transbordamento não é uma solução adequada para um sistema de alto desempenho. Inagami e Foley [50] propõem uma nova máquina, na qual os dados são novamente circulados no anel quando ocasionam transbordamento. Isto contudo aumenta o trânsito de fichas e apenas transfere o gargalo para outro lugar, não se podendo dizer que o problema tenha sido completamente resolvido. O verdadeiro problema está no modelo utilizado, que aloca espaço implicitamente para um dado quando, após feitas as comparações, descobre-se que seu parceiro ainda não chegou. Esta implementação necessita de memória associativa (ou pseudo-associativa), que é a causa de todos os males.

- *O pipeline é assíncrono.* A falta de um relógio global a todo o sistema provoca atrasos desnecessários na transmissão de dados de uma unidade para outra e introduz um elemento adicional de não determinismo na computação. Muitos estágios do *pipeline* são *buffers* de entrada e saída das unidades, sem função útil e apenas contribuindo para aumentar a latência

[1] speedup
[2] overflow

do anel. Isto significa que o tempo de propagação dos dados da entrada do anel para a saída é mais alto do que o necessário. Com um número tão grande de estágios, cerca de 40, observa-se que sob um baixo grau de paralelismo o pipeline não fica saturado por um longo período de tempo e ocorre uma séria degradação de desempenho [32].

- *Há trânsito excessivo de fichas no anel.* Este é devido em grande parte ao alto número de fichas destinadas a instruções que têm um único operando. Essas fichas, que não têm possibilidade de emparelhar e passam direto pela Unidade de Agrupamento, constituem em média 60% de todas as fichas que circulam no anel [42]. Uma proposta inicial tencionava solucionar este problema criando um caminho alternativo no anel para estas fichas, desviando-as da Unidade de Agrupamento. A idéia era que além de descongestionar a Unidade de Agrupamento e melhorar o seu desempenho, as bolhas causadas pela falha no agrupamento pudessem ser recuperadas. No entanto, simulações demonstraram que não se obtém ganhos reais consideráveis e atualmente acredita-se que a melhor abordagem é agrupar estas instruções em blocos que são executados seqüencialmente.

### 4.1.2 Pontos positivos

Apesar de terem sido apresentadas várias deficiências no projeto da MFDM, elas não inviabilizam a idéia de que uma arquitetura dirigida pelos dados é muito mais adequada à construção de sistemas multiprocessadores de alto desempenho do que o modelo tradicional von Neumann dirigido por controle. Embora não tenha conseguido obter um desempenho comparável ao dos supercomputadores, o que por sinal não era o seu objetivo, a MFDM conseguiu provar que o modelo de fluxos de dados é muito conveniente para o processamento, justamente por ser inerentemente paralelo.

A MFDM tratou de maneira conveniente as duas questões consideradas fundamentais no projeto de um multiprocessador: latência de memória e sincronização [10]. O modelo de fluxo de dados não possuindo o conceito de armazenamento, não existe o problema de os processadores ficarem ociosos esperando respostas da memória. Os processadores são constantemente alimentados com pacotes completos contendo as instruções e seus operandos, as quais são processadas como entidades auto contidas, sendo transparente a eles qualquer tipo de questão envolvendo memória.

A sincronização na MFDM, por ser uma das bases para a implementação do modelo de fluxo de dados, é realizada ao nível de instrução. Cada instrução pode

ser considerada como um processo muito leve, e os processos são sincronizados diretamente pelo hardware da máquina. Comparada com a sincronização nas máquinas von Neumann, a sincronização na MFDM é muito eficiente, muito embora a Unidade de Agrupamento seja lenta e tenha se tornado o principal gargalo do sistema. Mas esta deficiência é causada por uma decisão de projeto, que pode ser modificada para obter um maior desempenho sem destruir as características de sincronização de granularidade fina e reentrância de código do modelo de fluxo de dados dinâmico.

Muitos projetos de arquiteturas de fluxo de dados enfatizam a simplicidade dos anéis de processamento, necessitando de muitos processadores e poderosas redes de interconexão para obter um alto desempenho. O projeto da MFDM concentrou seus esforços em construir um anel bastante poderoso e, portanto, necessitando de redes muito menores para obter a mesma potência computacional [42]. Duas vantagens podem ser vistas nesta abordagem. Em primeiro lugar, as redes de interconexão atualmente representam um custo muito alto no projeto de um multiprocessador. Redes muito poderosas, como o *crossbar*, são evitadas por seu alto custo, dando lugar a redes mais simples, mas com latência mais alta. Em segundo lugar, o alto custo das transmissões na rede leva à distribuição da execução de blocos de código inteiros entre os anéis. Isto introduz a necessidade de balanceamento de carga, que além de nunca conseguir ser perfeito apresenta uma sobrecarga adicional. Na MFDM qualquer processador dentro de um anel pode executar qualquer instrução, o que representa uma utilização mais eficiente do poder computacional da máquina. Além disso, resultados de simulação da MFDM com vários anéis não demonstraram um aumento muito grande em desempenho [41], o que parece ser um grande indicador de que se deve reforçar ainda mais a capacidade de processamento dos anéis individuais.

O modelo de fluxo de dados e as linguagens de alto nível que o implementam representam um meio poderoso de representação de computação paralela. O projeto da MFDM, no entanto, excedeu-se ao divergir muito radicalmente das arquiteturas tradicionais von Neumann, que possuem um hardware muito eficiente para a execução seqüencial. Isto fez com que o recursos do hardware fossem mal utilizados, desperdiçando-se conhecimentos adquiridos em décadas de experiência com computadores digitais. Os projetos na área de fluxo de dados têm muito a aprender com os computadores von Neumann, como por exemplo, a execução de código seqüencial e o gerenciamento eficiente de memória. Na próxima seção será visto que a arquitetura básica da MFDM pode ser reformulada, sanando suas deficiências com várias características das arquiteturas von Neumann.

## 4.2 A Arquitetura proposta

A arquitetura da MX mantém as características mais importantes da MFDM, enquanto procura eliminar as suas deficiências através de técnicas já consolidadas em arquiteturas von Neumann e de novas idéias que vêm sendo propostas para arquiteturas híbridas. O projeto prevê uma velocidade de pico de 500 Mips como um objetivo imediato. Em especial, espera-se que esta máquina apresente um desempenho real próximo do desempenho previsto, para que haja um bom aproveitamento dos recursos de hardware e uma relação custo/desempenho melhor que a das máquinas disponíveis comercialmente no momento.

Dois princípios básicos fundamentam o projeto da MX:

(a) Qualquer processador pode executar qualquer instrução habilitada, ou seja, nenhum processador fica ocioso enquanto existe serviço para ser feito. Isto contribui para um melhor aproveitamento do potencial da máquina, pois desde que haja paralelismo suficiente no problema os processadores são plenamente utilizados. Esta é uma característica herdada da MFDM e que vai contra a maioria dos projetos de arquiteturas de fluxo de dados e de arquiteturas híbridas.

(b) A sincronização dos operandos não pode ocorrer no caminho crítico da execução, evitando o aparecimento de bolhas que em muitas máquinas, inclusive a MFDM, condenam os processadores a desperdiçarem vários ciclos sem trabalho útil. Esta é a maior contribuição retirada do projeto da MDFA, onde chama-se princípio *argument-fetching*.

A MX poder ser vista como a união de um multiprocessador de fluxo de dados com um multiprocessador convencional de memória compartilhada. Isto faz com que seja considerada uma arquitetura híbrida.

O modelo de armazenamento da MX é baseado em segmentos, sendo que um segmento é alocado implicitamente para receber os dados de um determinado bloco de código no momento da sua ativação. Na realidade, esse tratamento é análogo ao uso de registros de ativação nas arquiteturas híbridas, discutido no Capítulo 3. Para ter acesso aos dados, as instruções especificam endereços de memória, que são considerados deslocamentos em relação à base do segmento.

As fichas possuem a forma $< PS, PI >$, onde PS é o ponteiro para a base do segmento e PI o ponteiro para a instrução a ser executada. Ambos possuem 32 bits, formando assim uma ficha de 64 bits. No PI, 2 bits são utilizados para funções de sincronização, sendo um para indicar o número de operandos da ins-

trução (2 ou 1) e outro para indicar a porta de entrada (esquerda ou direita), no caso de instruções com dois operandos.

Instruções podem ser executadas seqüencialmente em blocos determinados estaticamente pelo compilador. Nesse caso, PI indica a primeira instrução do bloco, que é escalonado à maneira de fluxo de dados. As demais instruções do bloco são executadas em seqüência com escalonamento implícito von Neumann. Além de aumentar a eficiência na execução de código estritamente seqüencial e facilitar a implantação de gerenciamento de recursos na própria máquina, esta técnica reduz o tráfego de fichas na proporção do tamanho médio dos blocos. Assim, é possível diminuir as exigências dos mecanismos que realizam a sincronização, tornando mais eficiente o escalonamento dos blocos.

A arquitetura da MX é mostrada na Figura 4.1. O sistema é dividido em dois módulos principais, a Unidade de Escalonamento de Instruções (UEI) e a Unidade de Processamento de Instruções (UPI). A UEI realiza o escalonamento explícito das instruções à maneira de fluxo de dados, ou seja, uma instrução (a primeira de um bloco) só é habilitada para a execução quando seus operandos já estão disponíveis. A UPI é responsável pelo processamento dos blocos de instruções, que foram escalonados pela UEI, possuindo características comuns às arquiteturas von Neumann convencionais. A interface entre elas é bastante simples. A UPI produz fichas que não carregam dados, apenas indicam que eles estão disponíveis na memória. Estas fichas são enviadas para a UEI, onde são interpretadas como informações para sincronização. Quando um bloco de instruções está habilitado, a UEI envia para a UPI um pacote contendo PS, PI e mais dois endereços de operandos.

Para validar o primeiro princípio básico do projeto da MX, segundo o qual qualquer processador possa executar qualquer instrução, é necessário que a memória na UPI seja globalmente endereçável. Garantir que o sistema de memória seja capaz de suprir as necessidades dos processadores independentemente de latência é o principal objetivo deste trabalho. Deste modo, será dada maior ênfase à UPI e, em especial, ao projeto do sistema de memória.

### 4.2.1 Unidade de Processamento de Instruções

A Unidade de Processamento de Instruções tem seu princípio de funcionamento baseado no modelo von Neumann e sua estrutura é bastante similar à da arquitetura DFES [91]. DFES é uma arquitetura superescalar von Neumann que procura obter alto desempenho separando as fases de busca de operandos, execução e armazenamento de resultados. Desse modo, ela consegue afastar as restrições impostas pelo gargalo de von Neumann, dando ao sistema de memória a capa-

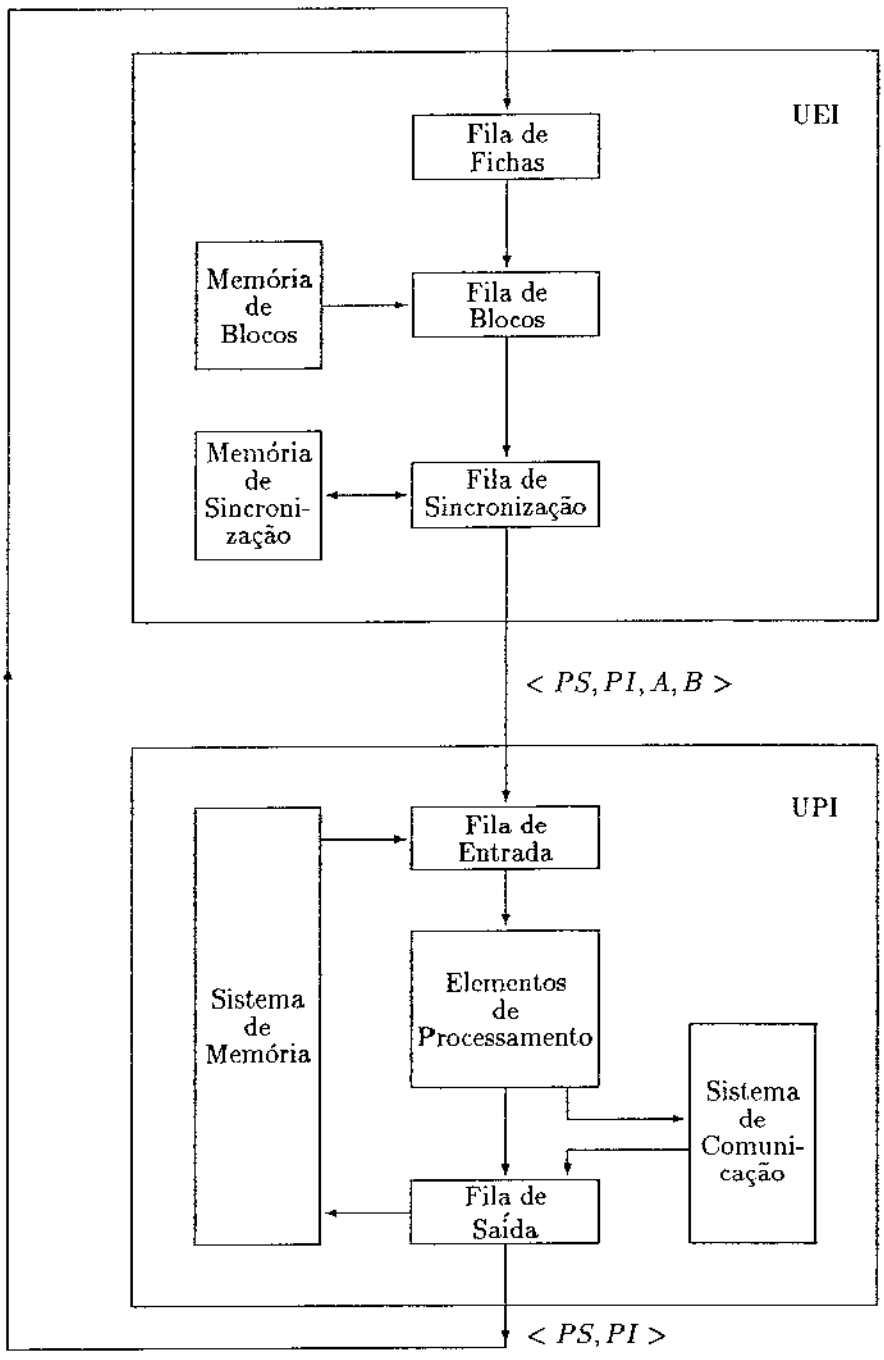



Figura 4.1: A arquitetura MX

cidade de suprir a necessidade de acesso rápido aos dados pelos processadores, evitando sua ociosidade. DFES pode ser enquadrada no conceito de *arquiteturas disjuntas*[3]. Tais arquiteturas têm apresentado bons resultados em estudos de simulação realizados, seja utilizando memória entrelaçada como apenas um módulo de memória [53].

A MX incorpora alguns princípios de DFES e os utiliza de maneira mais eficiente, na medida em que o escalonamento de instruções baseado em fluxo de dados a beneficia de duas maneiras. Em primeiro lugar não precisa respeitar a ordem estritamente seqüencial na execução das instruções adotada em DFES. Em segundo lugar, apresenta uma solução elegante para o escalonamento das instruções, uma questão tratada muito superficialmente em DFES. A seguir são apresentados com mais detalhes os componentes da UPI.

### Fila de Entrada

A Fila de Entrada recebe da UEI pacotes de 96 bits na forma $< PS, PI, A, B >$, onde A e B são endereços de 16 bits dos operandos na memória relativos a PS. Então envia requisições de leitura à memória, e quando os dados tornam-se disponíveis, libera pacotes executáveis de 192 bits aos Elementos de Processamento (EPs), com os operandos de 64 bits ocupando o lugar dos endereços.

Embora a estrutura lógica mais simples para a Fila de Entrada seja uma FIFO, como em DFES, esta imposição não é necessária na MX. Seu modelo de escalonamento garante que os blocos de instruções são livres de efeitos colaterais, de modo que blocos habilitados não precisam respeitar qualquer ordem de execução. Isto é benéfico uma vez que um sistema de memória para processamento paralelo é naturalmente baseado em algum esquema de memória entrelaçada, onde os conflitos nos acessos a módulos individuais podem gerar enormes perdas em desempenho. Na Fila de Entrada da MX os pacotes são enviados aos EPs tão logo estejam disponíveis os dados de que necessitam. Os conflitos nos acessos à memória podem ser em grande parte mascarados, não afetando o desempenho da máquina de maneira significativa.

A estrutura da Fila de Entrada é mostrada na Figura 4.2. Cada coluna representa um pacote acrescido de dois campos [A] e [B], para guardar os operandos de 64 bits. O número máximo de pacotes que a Fila de Entrada poderá conter será determinado com base em resultados experimentais.

[3] decoupled architectures

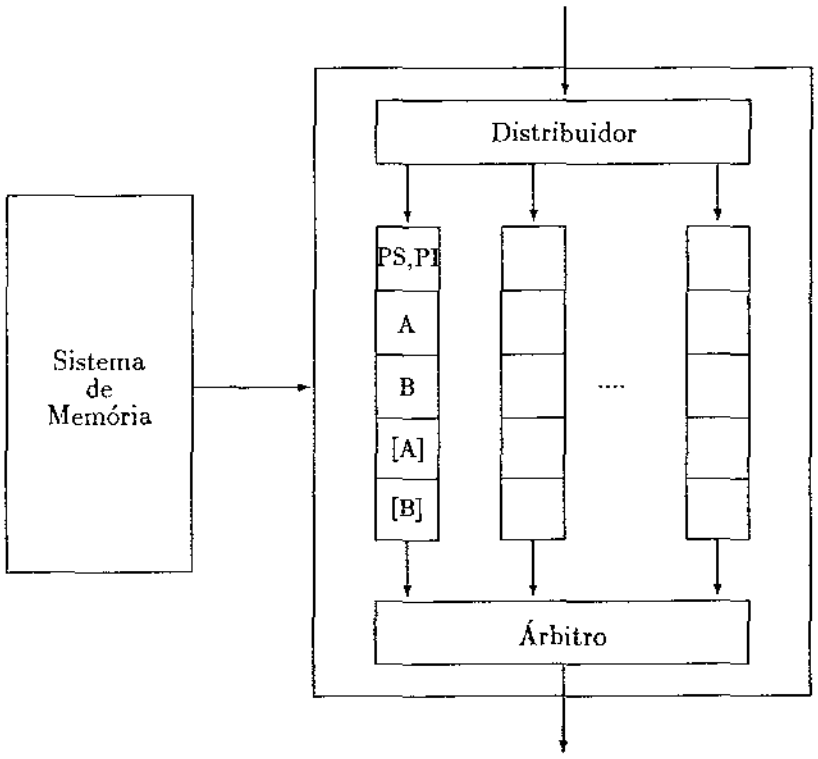



Figura 4.2: Fila de Entrada

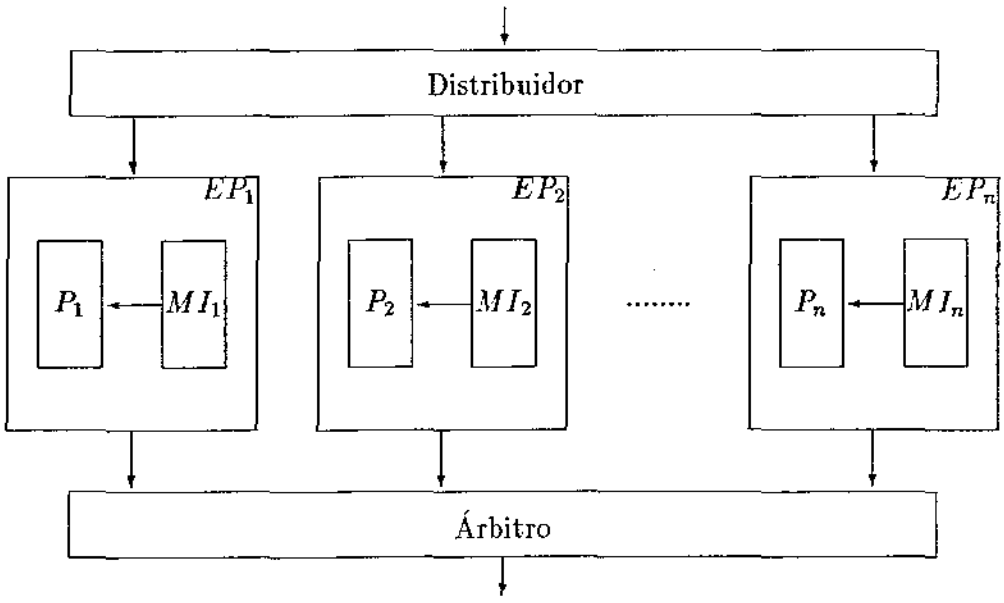



Figura 4.3: Elementos de Processamento

### Elementos de Processamento

Os Elementos de Processamento (EPs), cuja estrutura é mostrada na Figura 4.3, são os responsáveis pela execução das instruções na MX. Cada EP é composto de um processador com características RISC e de uma memória local para conter o código do programa executado. Os pacotes executáveis produzidos pela Fila de Entrada são delegados pelo distribuidor a qualquer EP disponível, sem restrições ou preferências. Durante a execução dos blocos de instruções, são eventualmente produzidos pacotes que são enviados à fila de saída pelo árbitro e indicam o armazenamento de dados e a criação de fichas.

Considerando processadores de 25 Mips, o que é compatível com a realidade do mercado atual, são necessários 20 processadores para atingir o desempenho desejado de 500 Mips. A opção feita no projeto da MFDM não é adequada a este caso, pois teria que ser utilizada uma quantidade grande demais de processadores microprogramáveis. Portanto, para a MX, são necessários processadores especialmente desenvolvidos, com características próprias.

A execução de um bloco de instruções nos EPs inicia-se com a chegada de um pacote executável. Os dois operandos, o PS e o PI são armazenados em registradores especiais, que são chamados respectivamente de RA, RB, RPS e RPI. O

RPI é utilizado para buscar a primeira instrução do bloco na memória. As demais instruções são executadas em seqüência, até que uma delas especifique o fim do bloco. Como os processadores não têm acesso direto à memória, as instruções somente podem especificar registradores como operandos, embora possam escrever seus resultados na memória, através da Fila de Saída.

Os processadores possuem um *pipeline* de instruções que é utilizado para aumentar a eficiência e encobrir o efeito da latência de memória. O *pipeline* não precisa ser esvaziado completamente a cada bloco executado, o que resultaria em uma perda em desempenho. Quando o término de um bloco é identificado nos primeiros estágios do *pipeline*, um outro bloco já pode ser iniciado.

### Sistema de Memória

O sistema de memória da MX pode ser dividido em três componentes básicos: módulos de memória, rede de interconexão e gerente da memória. Os módulos de memória são dispostos de maneira entrelaçada para permitir referências concorrentes a módulos diferentes. Cada módulo tem uma interface para a rede de interconexão, que por sua vez se comunica com as filas de entrada e saída. O gerente de memória é responsável por alocar, liberar e sincronizar segmentos e estruturas de dados. O sistema de memória é apresentado em detalhe na Seção 4.3.

### Fila de Saída

A Fila de Saída (Figura 4.4) tem uma estrutura bastante similar à da Fila de Entrada, mas com uma lógica mais simples. Ela recebe pacotes dos Elementos de Processamento na forma $< PS, PI, [C], C >$, onde PS e PI compõem a ficha a ser produzida e [C] é um valor a ser armazenado no endereço C. Este é o caso mais comum, quando os pacotes transportam dados a serem escritos nos módulos de memória. No caso de um pacote indicar uma requisição ao gerente de memória, os campos [C] e C contêm informações necessárias à operação desejada. Na realidade, a produção de fichas e a escrita de dados na memória não são necessariamente dependentes, podendo existir pacotes com uma ou outra função. Em última análise, a Fila de Saída pode ser vista como sendo formada por duas filas, uma para interface com a memória e outra para a emissão de fichas à UEI.

Ao ser recebido um pacote, uma ficha é produzida imediatamente e enviada à UEI, enquanto uma requisição de escrita é enviada ao sistema de memória. Não é necessário o reconhecimento por parte do sistema de memória, pois o tempo de sincronização na UEI é suficiente para que o dado seja escrito seguramente.

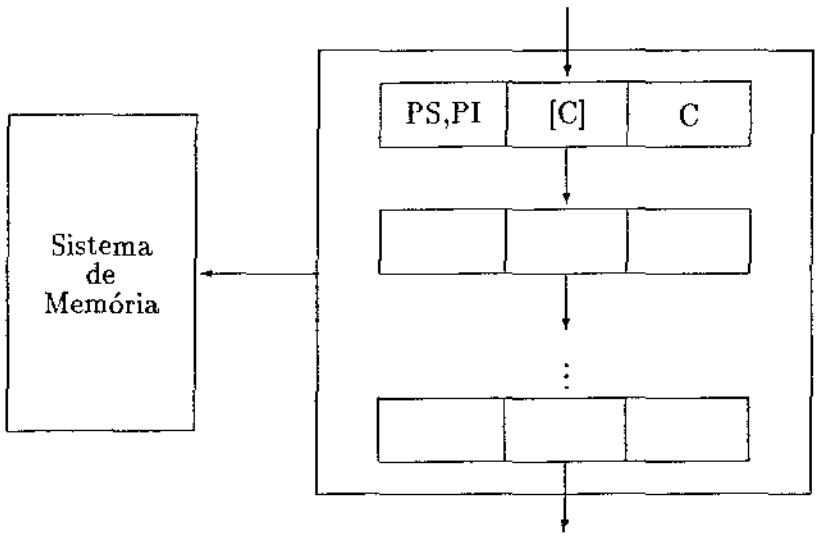



Figura 4.4: Fila de Saída

Também as requisições ao gerente da memória não precisam de uma resposta imediata. Desse modo, pode-se garantir que nunca irá ocorrer leitura de dados inconsistentes na memória.

Como o reconhecimento de escrita não é necessário para a criação de fichas e escrita de dados, a Fila de Saída pode ser uma FIFO, como em DFES. Isto torna sua implementação bem mais simples que a da Fila de Entrada.

### Sistema de Comunicação

O Sistema de Comunicação tem a função de interligar a MX com o mundo externo, ou seja, é uma unidade de entrada e saída para a comunicação com periféricos, com outros computadores, ou com outros sistemas idênticos à Figura 4.1, em uma configuração semelhante ao multianel da MFDM. A comunicação é realizada através de mensagens de 138 bits, formadas por um endereço para o dispositivo (10 bits), um contexto (64 bits) e um dado (64 bits). Mensagens de saída são geradas por instruções SAIDA (Seção 4.4.1) e enviadas ao Sistema de Comunicação, ao invés da Fila de Saída. Mensagens de entrada são recebidas de maneira assíncrona pela Fila de Saída e têm comportamento idêntico aos pacotes enviados pelos EPs, escrevendo dados na memória e produzindo fichas de habilitação.

A intenção do projeto é prover à MX um sistema operacional para que ela seja totalmente independente. No entanto, em estágios preliminares não se prevê a construção desse ambiente, de modo que um computador hospedeiro terá que ser utilizado para gerenciamento de recursos, como na MFDM. A comunicação com este computador hospedeiro também será feita através do sistema de comunicação. Sua arquitetura é uma rede de interconexão, cuja dimensão depende do número de dispositivos que serão conectados a ela.

### 4.2.2 Unidade de Escalonamento de Instruções

A Unidade de Escalonamento de Instruções (UEI) tem a função de apresentar à UPI as instruções (ou blocos de instruções) habilitadas à execução. Sua função é idêntica à ISU da MFDA, embora elas utilizem métodos de escalonamento distintos. O objetivo da existência de uma unidade separada para o escalonamento das instruções é eliminar as bolhas formadas quando as fichas não encontrarem suas parceiras no momento da sincronização. Estas bolhas são inevitáveis na MFDM e em outros projetos, como DFVN, P-RISC e Monsoon. Na MX, desde que haja suficiente paralelismo e que a UEI consiga um bom desempenho no tratamento das fichas, o impacto negativo das sincronizações mal sucedidas pode ser totalmente escondido da UPI.

#### Fila de Fichas

A Fila de Fichas tem uma função similar à da MFDM armazenando fichas temporariamente para abrandar os desníveis provocados pelas taxas irregulares de produção na UPI e de consumo na UEI. Sua estrutura lógica é uma FIFO e ela deve ser capaz de suprir as demandas da UPI e da UEI. Seu desempenho deve ser tal que o resto do sistema praticamente não perceba sua existência, isto é, não pode ser a causadora de quaisquer atrasos. A implementação proposta por Inagami e Foley [50] pode ser tomada como base para o projeto da Fila de Fichas da MX, pois leva em consideração várias restrições de tempo e consegue obter um desempenho aproximado de 30 Gb/s, através da utilização de memória entrelaçada.

#### Memória de Blocos

A Memória de Blocos armazena informações relacionadas com a sincronização e os operandos dos blocos de instruções. As posições da Memória de Blocos são endereçadas por PI e contêm *indicadores de bloco* de 48 bits, que possuem a forma $< A, B, E >$, onde E é um endereço de sincronização e A e B são

endereços dos operandos do bloco na UPI, todos com 16 bits. Os indicadores de bloco são gerados pelo compilador e os seus endereços são os mesmos da primeira instrução dos blocos na UPI. Por este motivo, a menos que todos os blocos possuam tamanho unitário, várias posições da Memória de Blocos ficarão desocupadas, pois ela deve ter o mesmo tamanho físico da memória de instruções dos EPs na UPI, para que o mapeamento seja correto.

Para suprir as necessidades do sistema, o desempenho da Memória de Blocos deve ser alto, o que leva à utilização de memória entrelaçada e de uma interface similar à Fila de Entrada da UPI, a Fila de Blocos. Na realidade, a implementação da Memória de Blocos e da Fila de Blocos pode ser bastante simplificada. Como as informações contidas são estáticas, a Memória de Blocos pode ser replicada. Isto representa custo extra em componentes, mas reduz drasticamente a complexidade na implementação da memória entrelaçada e da Fila de Blocos.

A Memória de Blocos recebe fichas constantemente da Fila de Fichas e envia à Memória de Sincronização pacotes de 112 bits na forma $< PS, PI, A, B, E >$.

### Memória de Sincronização

A Memória de Sincronização realiza o emparelhamento dos operandos destinados a um mesmo bloco de instruções. As posições da Memória de Sincronização contêm apenas um bit que indica presença ou ausência do operando. Se o bit indicar ausência, o pacote é destruído e o bit passa a indicar presença. Se o bit indicar presença, o pacote, menos o endereço E, é enviado à UPI. Além disso, o conteúdo do bit é alterado novamente para ausência, para posterior utilização, do mesmo modo que ocorre em Monsoon.

Quando um pacote é destruído, no caso de ausência do operando, uma bolha é gerada no sistema. A questão então é conseguir um desempenho alto na UEI, de modo que as bolhas possam ser escondidas da UPI. A solução é a realização de várias sincronizações simultaneamente, através da utilização de memória entrelaçada. Desse modo, um pacote habilitado pode sempre ser inserido no lugar de uma bolha. A Fila de Sincronização possui um comportamento similar à Fila de Entrada da UPI, como a Fila de Blocos, para que seja aliviado o impacto das colisões nos módulos de memória entrelaçada. Assim como a Memória de Blocos, a Memória de Sincronização pode ser replicada para facilitar a implementação, embora isto acarrete problemas com a distribuição da sincronização.

O endereçamento na Memória de Sincronização é relativo a PS, ou seja, E é um deslocamento a partir da base do segmento PS. Existe uma relação direta entre os segmentos de dados na UPI e na Memória de Sincronização. Quando um bloco de código é invocado, o gerente da memória aloca áreas tanto na memória

de dados da UPI, quanto na Memória de Sincronização da UEI, ambas apontadas por PS.

## 4.3 Sistema de Memória

O Sistema de Memória pode ser considerado a parte mais crítica da MX, e o seu projeto é o alvo primordial deste trabalho. A necessidade do armazenamento de dados em uma máquina de fluxo de dados como a MFDM já foi descrito anteriormente. No entanto, para que possam ser obtidos benefícios dessa abordagem, deve-se tomar o cuidado extremo de não tornar o Sistema de Memória um gargalo da máquina. Um dos problemas fundamentais com que os processadores von Neumann se vêem envolvidos quando conectados em um multiprocessador é a excessiva latência nos acessos à memória. Isto causa uma degradação muito grande do desempenho, a qual aumenta na medida proporção em que mais processadores são acoplados à máquina. Então, torna-se imprescindível que para o projeto da MX sejam obtidos todos os benefícios das máquinas von Neumann, aliados à ausência de latência em acessos à memória das máquinas de fluxo de dados.

Inicialmente havia duas possíveis soluções para a ligação dos processadores à memória, ambas baseadas em processadores von Neumann convencionais: memória compartilhada ou memória distribuída. Nenhuma das duas ajustou-se ao projeto da MX, por apresentarem desvantagens inaceitáveis. Com os processadores tendo acesso a uma memória compartilhada através de uma rede de interconexão, há um retorno aos problemas das máquinas von Neumann, com tempos de latência muito elevados. Em um sistema de memória distribuída, tendo cada processador acesso privado a uma porção da memória, este problema pode ser bastante amenizado. No entanto, essa solução contraria o primeiro princípio básico do projeto da MX, uma vez que distribui antecipadamente a execução das instruções a processadores específicos.

A solução encontrada foi impedir o acesso dos processadores diretamente à memória (Figura 4.1), havendo um estágio inicial para a leitura dos operandos (Fila de Entrada) e um estágio final para a escrita dos resultados (Fila de Saída). Desse modo, para os processadores a existência da memória é transparente, recebendo pacotes prontos para a execução, como na MFDM. Do ponto de vista da UPI, tem-se o efeito de uma memória compartilhada. No entanto, para que a existência da memória não interfira no desempenho global da máquina, ela não pode causar nenhum atraso, ou seja, tem que ser capaz de suprir as necessidades dos processadores, de modo que eles nunca se tornem ociosos enquanto houver

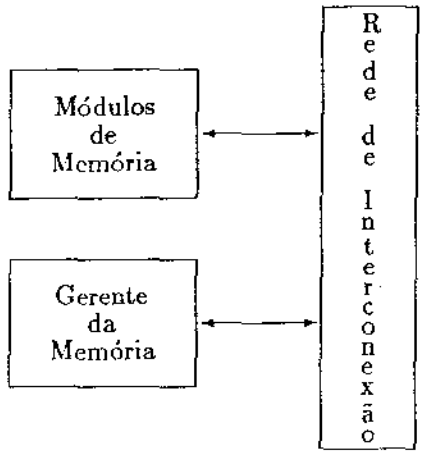



Figura 4.5: O Sistema de Memória

trabalho a ser feito. Para que o desempenho esperado de 500 MIPS seja atingido, o sistema de memória deve ter um desempenho da ordem de alguns Gb/s. Este objetivo é possível de ser alcançado e esta questão é tratada com mais detalhes no Capítulo 5.

A Figura 4.5 mostra a arquitetura do Sistema de Memória da MX. Ele é composto de Módulos de Memória, Rede de Interconexão e Gerente da Memória, apresentados a seguir.

### 4.3.1 Módulos de Memória

Para obter o alto desempenho necessário, é utilizado um esquema de memória entrelaçada, ilustrado na Figura 4.6. No Sistema de Memória estão localizados vários Módulos de Memória, cada um possuindo o seu próprio controlador de memória (CM). Nesta seção, para fins de simplicidade assume-se que o espaço de endereçamento é distribuído uniformemente entre os M módulos de memória em ordem linear. Isto é, o i-ésimo módulo ($0 \leq i < M$) guarda os endereços i, M + i, 2M + 1, ... O tamanho da palavra armazenada em cada módulo é de 64 bits, uma vez que a MX destina-se principalmente a resolver problemas científicos e deve ser capaz de tratar de operações de ponto flutuante de precisão dupla diretamente ao nível de hardware.

O controlador de memória é composto de uma interface para a rede de interconexão, uma fila para requisições e resultados e uma interface para a memória. A existência da fila torna o sistema mais eficiente no tratamento de conflitos, permitindo obter maior benefício do entrelaçamento de memória. Como a memória

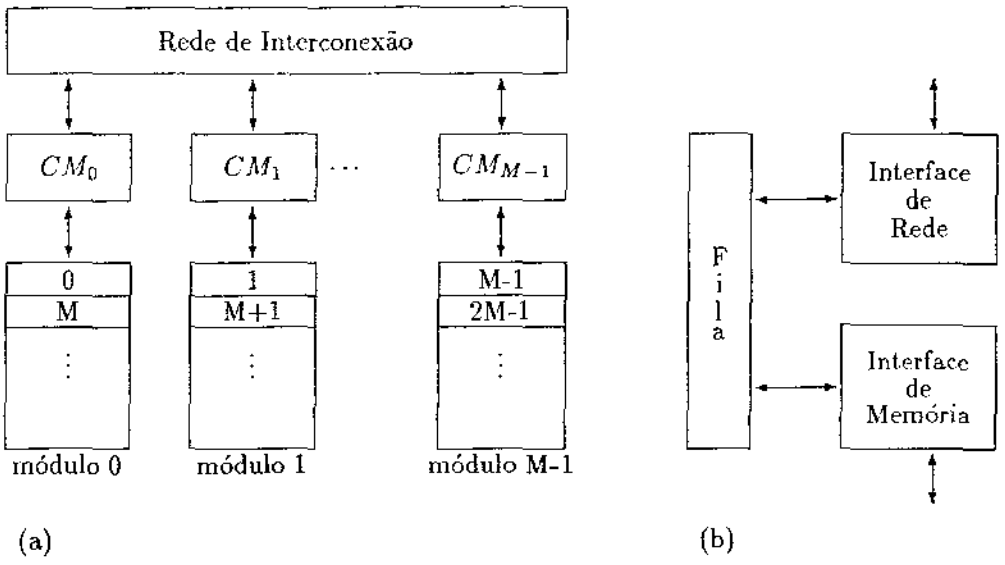



Figura 4.6: Detalhamento do Sistema de Memória: (a) Módulos de Memória; (b) Controlador de Memória (CM)

física somente pode processar uma requisição por vez, deve haver algum dispositivo para armazenar temporariamente as requisições, ao invés de simplesmente rejeitá-las quando a memória está ocupada. A fila também serve para sincronizar a interface de rede e a interface de memória, ou seja, para que a interface de rede não seja bloqueada quando a memória estiver ocupada e a memória não desperdice tempo esperando por requisições. A interface de rede é responsável por receber as requisições e preparar a devolução da resposta no caso de uma leitura. Além disso, deve sinalizar os elementos da fila, indicando requisições de leitura ou escrita para a interface de memória. A interface de memória monitora os elementos da fila e o seu status (leitura ou escrita), transfere as requisições para a memória, e, no caso de leitura, os resultados de volta para a fila.

De acordo com a definição apresentada em [47], a memória entrelaçada da MX segue o esquema C, onde os acessos aos módulos de memória podem ser efetuados concorrentemente. O esquema S (acesso simultâneo), embora de implementação mais simples, não é aceitável para esta situação, sendo mais adequado ao processamento vetorial, onde os dados são mais freqüentemente tratados em seqüência. Com a memória entrelaçada C e a fila interna ao controlador de memória, pretende-se alcançar todos os benefícios dos M módulos de memória.

### 4.3.2 Rede de Interconexão

O acesso rápido aos dados traz a necessidade de que a Fila de Entrada e a Fila de Saída sejam ligadas aos módulos de memória através de uma rede de interconexão que não introduza muitos ciclos de espera adicionais. Além disso, a rede de interconexão deve ter um alto *bandwidth*, para dar vazão à alta taxa de dados que o sistema necessita. Em princípio, não se está considerando a possibilidade de utilizar redes multiestágio, ou qualquer outra que possua fases intermediárias entre os pontos de comunicação. O *crossbar* seria a rede mais indicada para este caso, por apresentar o menor atraso possível. No entanto, seu alto custo afasta a possibilidade da sua utilização, pois cada posição das filas de entrada e saída deve ter um acesso direto a cada módulo de memória. Do mesmo modo, a utilização de um barramento simples também não é possível, uma vez que a demanda por transmissão de informações imposta pela MX vai além da sua capacidade.

Em DFES, que possui uma arquitetura semelhante à UPI da MX, a solução encontrada foi a utilização de múltiplos barramentos. O controle é distribuído, ou seja, a memória não centraliza o controle sobre as requisições que estão sendo processadas e cada elemento que efetua a leitura de um dado fica constantemente monitorando os barramentos para identificar se ele já está disponível. O controle é atribuído aos módulos de memória e aos elementos que emitem as requisições

de leitura e escrita.

Como são empregados múltiplos barramentos, é necessário um algoritmo arbitrador eficiente, para impedir situações de *deadlock* e *starvation*. Em [91] podem ser encontrados o algoritmo arbitrador e a prova de que ele é capaz de tratar com os problemas tradicionais de alocação concorrente de múltiplos recursos: corretude, segurança, justiça e degradação graciosa. DFES também utiliza um protocolo de barramento de barramento síncrono de 4 fases (arbitração, transferência da informação, espera e reconhecimento), que garante a transferência das informações sem distorções ou perdas. O protocolo possui a capacidade de *pipelining*, de modo que, uma vez ocupadas todas as fases, a cada ciclo de barramento é transmitida ma informação.

Este esquema de múltiplos barramentos de DFES pode ser adaptado à MX praticamente sem modificações. Deve-se levar em consideração as diferenças na estrutura das filas de entrada e saída nos dois projetos e a existência do Gerente da Memória na MX, o qual também deve ser ligado à rede de interconexão.

### 4.3.3 Gerente da Memória

O Gerente da Memória tem a função de alocar e liberar segmentos e estruturas de dados. Estruturas de dados são tratadas como segmentos e o acesso a seus elementos é feito a partir do seu endereço inicial, representado pelo ponteiro de segmento (PS). A arquitetura do Gerente da Memória é baseada na Sub-unidade de Alocação da Unidade de Estruturas da MFDM [71]. Requisições para alocar espaço chegam ao gerente contendo o tamanho do segmento (ou estrutura), o contexto de retorno (PS,PI) e um endereço na memória onde será guardado o novo PS. Como existe a limitação de somente dois operandos por instrução, foram consideradas duas opções para o envio deste endereço: embutí-lo no contexto de retorno de acordo com alguma regra pré-estabelecida, ou partilhar um dos operandos de 64 bits com o tamanho do segmento, cabendo a cada um 32 bits (Seção 6.1.1). Quando houver espaço disponível na memória, o gerente envia uma ficha ao contexto de retorno e escreve o novo PS na posição de memória especificada. Requisições para liberar segmentos contêm apenas o PS e não necessitam de resposta.

Estruturas de dados na MX podem ser sincronizantes ou não sincronizantes. Estruturas não sincronizantes têm um comportamento idêntico às estruturas de dados utilizadas nas máquinas von Neumann. Já as estruturas sincronizantes, como no caso das estruturas da MFDM, não podem sofrer acesso enquanto não houver uma confirmação explícita de que seus elementos foram produzidos. Na MX, em princípio, somente é possível a utilização de estruturas sincronizantes

estritas, ou seja, um elemento somente pode ser consumido quando todos os elementos da estrutura estiverem disponíveis. Esta escolha teve como base a simplicidade do projeto e não implica na impossibilidade da implementação de outros tipos de estruturas de dados na MX.

Quando uma estrutura é alocada, um contador interno é implicitamente atribuído a ela para fins de sincronização global[4]. A diferença entre estruturas sincronizantes e não sincronizantes reside no fato de que para as primeiras é necessário o envio de uma requisição de sincronização global, que contém o PS da estrutura e um contexto de retorno. Todas as requisições de escrita a estruturas são recebidas também pelo gerente da memória, para que possa ser efetuada a sincronização. Ao notar que uma estrutura sincronizante está pronta, o gerente envia uma ficha ao contexto de retorno, e então ela pode ser consumida.

Em Monsoon, o gerenciamento de memória foi implementado por funções especiais de software. Nas avaliações observou-se que cerca de 50% das instruções processadas se destinavam a este serviço. Na MX, foi feita a opção pela implementação do Gerente da Memória em hardware, aproveitando o projeto da Unidade de Estruturas da MFDM. Esta abordagem, apesar de incorrer em custo extra em componentes, facilita a utilização de segmentos (e estruturas) de tamanho variável. Na realidade, define-se uma quantidade mínima de espaço que pode ser alocada, e o tamanho de um segmento é sempre um múltiplo dessa quantidade. Essa quantidade é $2^p$ palavras de 64 bits, onde $p$ é o número de bits mais à direita em PS, que são desconsiderados para fins de endereçamento indireto. Isto permite que um endereço de memória seja embutido em PS, que pode ser utilizado na requisição para alocações de segmento e na instrução ENVIA (Seção 4.4.1). Como $p$ é um número pequeno, entre 3 e 6, a ser definido, essa quantidade mínima de alocação não deverá gerar desperdícios de memória.

A implementação do Gerente da Memória em hardware beneficia em certos aspectos o projeto do Sistema de Memória. Apesar de sua estrutura lógica ser uma unidade só, fisicamente pode-se distribuir o serviço a várias unidades para aumentar o desempenho e permitir maior concorrência. Para a Unidade de Estruturas da MFDM foi realizado um extenso estudo a esse respeito [51].

## 4.4 Aspectos de Software

A parte mais importante deste trabalho é o projeto da arquitetura da MX e, em especial, do Sistema de Memória. No entanto, para que o projeto seja mais completo, são necessárias algumas considerações a respeito do software e da ma-

[4] collect

neira como ele será executado nesta máquina. Questões mais genéricas, como a escolha de uma linguagem de programação de alto nível, a construção de um compilador e de um sistema operacional, não foram estudadas mais detalhadamente, dadas as sérias restrições de tempo a que está submetido este trabalho. Sobre isso são apenas feitas algumas discussões superficiais, apresentadas como trabalhos futuros no Capítulo 6.

Nesta seção serão tratadas apenas as questões mais diretamente relacionadas à arquitetura, como o conjunto de instruções, o protocolo de chamada e retorno de procedimento e particularidades da execução de instruções sobre estruturas de dados e vetores. Em geral, todos os aspectos relacionados a software são de alguma forma baseados no projeto da MFDM, embora fortemente influenciados pelos projetos atuais de arquiteturas híbridas. O conjunto de instruções, por exemplo, foi bastante modificado em relação à MFDM, tendo-se optado por um conjunto reduzido de instruções, como será visto a seguir.

### 4.4.1 Conjunto de instruções

A MX possui um conjunto reduzido de instruções, ao estilo dos novos projetos de arquiteturas híbridas, como P-RISC e EM-4, e também dos supercomputadores da linha NEC [56], cujo processador escalar segue a filosofia RISC. Instruções possuem um tamanho máximo de 64 bits, na forma $< OP, A, B, C >$, onde OP é o código de operação e A, B e C são registradores, constantes ou endereços.

Três modos de endereçamento são permitidos, sendo o mais comum deles o endereçamento de registrador, muito utilizado pelas instruções aritméticas, lógicas e de desvio. O endereçamento imediato também é permitido, com a possibilidade de constantes e endereços de dados ou instruções serem codificados como parte das instruções. O endereçamento base-deslocamento é realizado em todo acesso à memória, pois os endereços são sempre relativos a algum segmento (PS). Este modo de endereçamento somente pode ser utilizado na instrução STORE e, de maneira mais restrita, na instrução ALOCA-SEGMENTO. Além destes, o endereçamento indireto é utilizado, mas de maneira pouco convencional, limitado ao tratamento de vetores (Seção 4.4.3), ou seja, não é disponível ao programador em toda a sua generalidade.

Os tipos de dados suportados diretamente pela arquitetura da MX são inteiros em complemento de dois de 64 e 32 bits, inteiros sem sinal de 64 e 32 bits, números de ponto flutuante de precisão dupla (64 bits) e precisão simples (32 bits) e valores lógicos de 32 bits. Além desses, são necessários tipos de dados especiais para funções de controle, como *contexto* (PS,PI) de 64 bits, *ponteiro de instrução* (PI) de 32 bits, *ponteiro de segmento* (PS) de 32 bits e *endereço*

*relativo* de 16 bits. Ao contrário do que ocorre na MFDM [41], a máquina confia a tarefa de verificação de tipos às linguagens de programação de alto nível nela implementadas. Dessa forma, dispensa os bits identificadores de tipo utilizados na MFDM e permite o acesso a valores menores do que a palavra de dados de 64 bits.

As instruções da MX podem ser classificadas em três grupos principais:

(a) instruções aritméticas, lógicas e de desvio;

(b) instruções de ponto flutuante;

(c) instruções de controle.

O primeiro grupo é formado por instruções RISC convencionais, embora as instruções de desvio sejam baseadas na MFDM. Podem endereçar registradores e constantes inteiras ou reais de 32 bits.

Para a implementação das instruções de ponto flutuante de precisão dupla, mais complexas e, portanto, de execução mais lenta, foram consideradas 3 opções:

(a) os processadores as executam normalmente;

(b) instruções complexas são divididas em duas ou mais instruções simples que são executadas seqüencialmente em um bloco, como uma macroinstrução;

(c) a cada processador é associado um coprocessador aritmético, que executa as instruções de ponto flutuante.

A opção 1 é utilizada na MFDM, onde a grande latência do anel compensa a execução mais lenta. Além disso, o fato de os processadores serem microprogramáveis facilita sua implementação. Na MX implicaria em um desvio da filosofia RISC. As opções 2 e 3 merecem um estudo mais atencioso. A opção 2 favorece a implementação de outras instruções complexas, como algumas instruções de controle, mas a opção 3 representa maior velocidade de processamento. Na realidade, a maioria das arquiteturas RISC utiliza coprocessadores aritméticos para a execução de instruções de ponto flutuante, de modo que este é o caminho mais provável a ser seguido. Apesar disso, a opção 2 não deve ser descartada, pois representa uma ótima possibilidade para a implementação de instruções complexas a partir de instruções mais simples.

As instruções de controle são utilizadas para acesso à memória, produção de fichas, comunicação entre procedimentos e entrada e saída e emissão de requisições ao gerente da memória. Elas são apresentadas mais detalhadamente a seguir:

- STORE (Str) *arg, end, ps*: gera uma requisição para escrever *arg* no endereço absoluto *ps* + *end* da memória. Se *ps* for omitido, usa-se o registrador RPS, que armazena o PS do segmento atual. Não existe instrução de leitura de memória (LOAD), pois esta operação é realizada implicitamente pela Fila de Entrada. De certa forma, a MX é semelhante às arquiteturas LOAD/STORE [26], incorporadas pelos processadores RISC. A maior diferença reside no fato de que na MX o armazenamento explícito de dados é utilizado somente para aumentar a sua eficiência, não fazendo parte do seu modelo. Devido à influência do modelo de fluxo de dados, conceitualmente não existe a ligação indesejável, porém inevitável, entre variável e posição de memória como nas máquinas von Neumann.

- FORMA-CONTEXTO (Fct) *ps, pi, contexto*: agrupa *ps* e *pi* para formar *contexto*. É utilizada como preparação para a instrução ENVIA.

- ENVIA (Snd) *contexto, n*: implementa uma aresta dinâmica para a comunicação entre blocos de código, produzindo uma ficha indicada em *contexto*, na forma $< PS, PI + n >$. ENVIA é na realidade a primeira parte de uma instrução complexa e sempre deve estar seguida de uma instrução STORE *arg, n, ps*, onde *arg* é o dado a ser transmitido e *ps* é igual ao PS de *contexto*. Ela poderia ser implementada como uma única instrução, porém foi feita a opção por duas instruções mais simples.

- SINALIZA (Sgn) *pi*: produz uma ficha na forma $< RPS, pi >$, sendo utilizada para comunicação (habilitação) entre blocos de instruções. Seqüências dessas instruções substituem árvores de instruções DUP, que na MFDM consomem grande parte do tempo de processamento dos programas [42]. Freqüentemente faz-se necessário escrever um dado na memória e produzir uma ficha habilitando uma instrução a consumí-lo. Este é outro caso onde seria desejável uma instrução complexa, mas a utilização conjunta das instruções STORE e SINALIZA representa uma solução mais simples.

- ALOCA-SEGMENTO (Als) *tam, contexto, end*: produz uma requisição ao gerente da memória para alocar um segmento de tamanho *tam*, cujo PS é escrito no endereço absoluto de memória RPS + *end*. Uma ficha é enviada ao contexto de retorno *contexto* quando o segmento estiver disponível.

- LIBERA-SEGMENTO (Rls) *ps*: produz uma requisição ao gerente da memória para liberar o espaço de memória ocupado pelo segmento apontado por *ps*.

- SINCRONIZA (Syn) *ps, contexto*: produz uma requisição ao gerente da memória para realizar uma sincronização global em uma estrutura de dados sincronizante e enviar uma ficha ao contexto de retorno *contexto* quando a estrutura estiver completa.

- SAIDA (Out) *disp, contexto, arg*: produz uma mensagem ao sistema de comunicação, endereçada ao dispositivo *disp* com o argumento *arg*; *contexto* pode conter o contexto de retorno à MX, ou uma identificação mais detalhada sobre o dispositivo e a operação a ser realizada. Em geral, a operação desejada é indicada em *disp*.

### 4.4.2 Chamada e retorno de procedimento

O modelo de fluxo de dados dinâmico permite que múltiplas ativações de um determinado vértice (instrução) estejam habilitadas, ou parcialmente habilitadas, simultaneamente. Diferentes ativações são identificadas pelos rótulos carregados nas fichas. Na MX esta diferença é especificada por PS, indicando recursos diferentes, no caso segmentos de memória, onde os dados de uma particular ativação de um bloco de código são armazenados.

De uma maneira simplificada o protocolo de chamada e retorno de um procedimento na MX pode ser dividido em cinco fases: alocação de um segmento, envio dos argumentos, execução do procedimento, recebimento dos resultados e liberação do segmento. Na realidade, porém, são necessários maiores cuidados. A Figura 4.7 mostra, em forma de grafo, um exemplo da convenção de chamada e retorno de procedimento, onde o procedimento P invoca o procedimento Q, que possui dois argumentos ($x$ e $y$). As arestas do grafo não indicam o fluxo de dados, como está sugerido na figura, mas somente o fluxo de controle. A figura está representada como um grafo para facilitar a visualização, e muitas simplificações foram feitas, como, por exemplo, a omissão de algumas instruções STORE e SINALIZA que seriam necessárias.

O procedimento P primeiramente aloca um segmento para o procedimento Q, através da instrução ALOCA-SEGMENTO, que é uma operação multifase. Duas instruções FORMA-CONTEXTO são utilizadas, uma para produzir o contexto de entrada no procedimento Q e a outra o contexto de retorno ao procedimento P. Instruções ENVIA + $i$ são empregadas para comunicar os argumentos e o contexto de retorno para Q. Na realidade, uma instrução ENVIA + $n$ é implementada como uma instrução ENVIA seguida de uma instrução STORE, onde $n$ representa um deslocamento para calcular o novo PI e o endereço em relação a PS, onde o dado será armazenado. Para retornar o resultado, o procedimento Q necessita de um

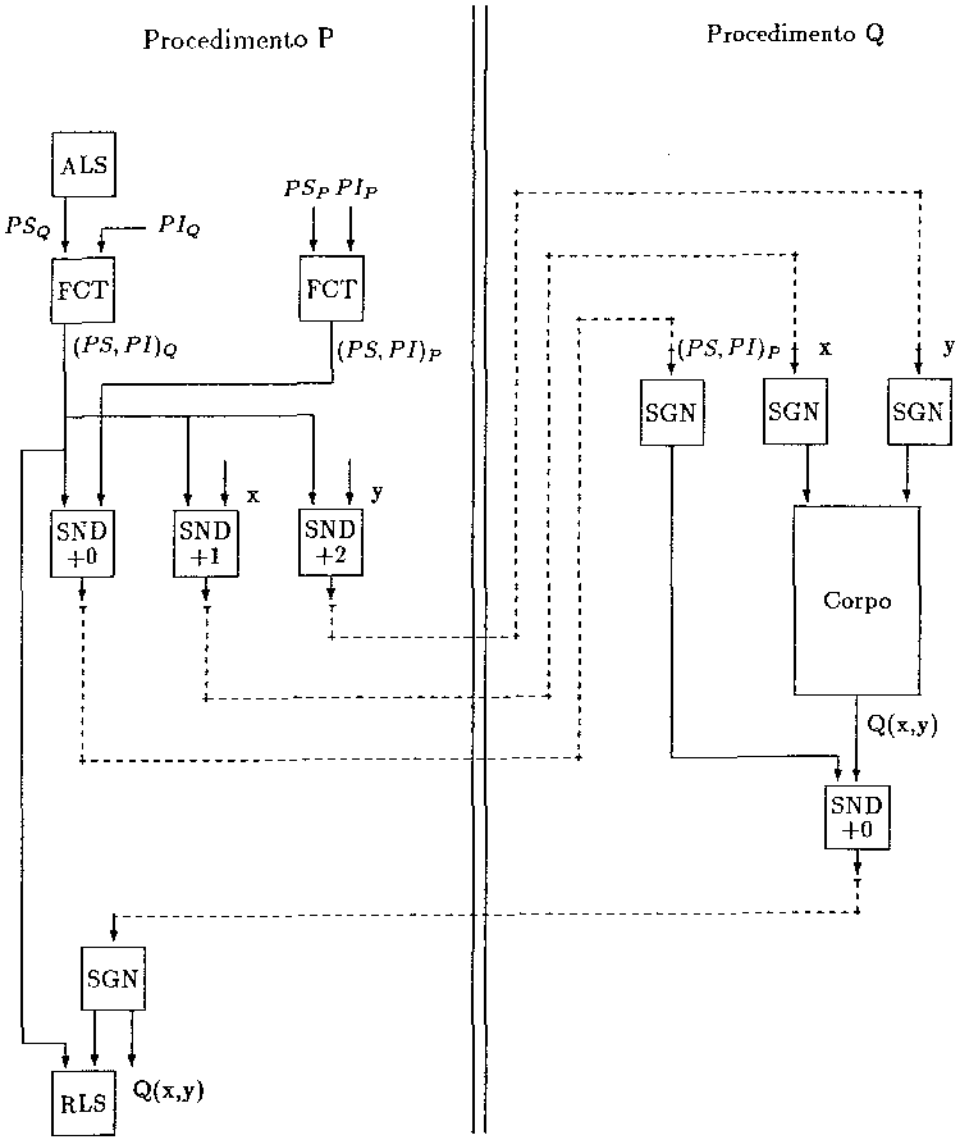



Figura 4.7: Exemplo de chamada e retorno de procedimento

endereço, que é a posição dentro do segmento de P onde ele deve ser escrito. No envio dos argumentos este endereço é obtido através de $n$, mas a mesma idéia não pode ser aplicada para o retorno do resultado. Então, esse endereço é embutido no PS do contexto de retorno (vide Seção 4.3.3), de modo que não é necessário enviar mais um argumento somente com ele, e o retorno do resultado pode ser realizado do mesmo modo que o envio dos argumentos. As arestas de saída das instruções ENVIA + $i$ são tracejadas na figura para representar que elas são geradas dinamicamente.

Na entrada do procedimento Q são colocadas instruções SINALIZA para que a interface entre os dois procedimentos seja estruturada. Os pontos de entrada de um procedimento são sempre as primeiras instruções do código e as primeiras posições do segmento, e o processo de compilação torna-se mais independente. Um processo de compilação menos estruturado poderia prescindir da utilização dessas instruções, otimizando o código.

A julgar pela Figura 4.7, somente o recebimento do resultado é necessário para liberar um segmento. Na prática, deve-se ter a certeza de que outras instruções cujos resultados não serão aproveitados, mas que possivelmente ainda estão em execução, já foram terminadas. Em ETS, o compilador inclui arestas de sinalização para detectar que a ativação está completa e o segmento pode ser liberado [59]. Em [20] pode ser encontrada uma sugestão similar a esta para a MFDM.

### 4.4.3 Resolução de indireções e instruções vetoriais

Um dos maiores problemas encontrados no início do projeto desta nova arquitetura foi a resolução de indireções, ou seja, o endereçamento indireto resultante da utilização de vetores ou matrizes. Em uma instrução que referencie o elemento $i$ do vetor $a$ (ou seja, a[i]), é impossível prever em tempo de compilação a exata posição da memória que ele irá ocupar. Isto ocorre pela característica dinâmica tanto da alocação de espaço para estruturas de dados, quanto da geração de índices. Não se sabe nem o endereço do início da estrutura, que é o PS, nem o deslocamento $i$ do elemento em questão.

O endereço efetivo de a[i] somente poderá ser conhecido em tempo de execução, e o endereço gerado pelo compilador armazenará este endereço dinâmico, portanto será uma indireção. Em máquinas von Neumann, os processadores possuem acesso direto à memória e podem efetuar quantas leituras forem necessárias, até descobrirem o endereço efetivo do dado que desejam. Na MX isto não é possível, uma vez que os processadores já recebem os pacotes prontos para a execução e não podem efetuar leituras na memória. A solução encontrada

foi deixar a resolução das indireções a cargo das filas de entrada e saída. Ao identificar que um endereço é indireto (pode-se utilizar um bit para isto), a fila de entrada não despacha o pacote quando recebe o resultado, mas efetua outra leitura para obter o elemento desejado. Já a fila de saída deve ser um pouco alterada para realizar esta tarefa. Um endereço indireto faz com que seja produzida uma requisição de leitura, para obter o endereço efetivo onde o elemento deve ser escrito. Desse modo, ela passa a se comportar como um dispositivo não somente de escrita, mas também de leitura.

Uma possível solução para a execução eficiente de instruções com indireções é a utilização de instruções vetoriais. O objetivo inicial era aumentar o desempenho de uma máquina de fluxo de dados com a inclusão de processadores vetoriais, que executam operações sobre vetores de maneira muito eficiente. No entanto, processadores vetoriais teriam que ter acesso direto à memória, o que aumentaria ainda mais a demanda por desempenho do sistema de memória. O efeito que isto poderia ocasionar na MX não é fácil de prever, mas é provável que houvesse um desbalanceamento entre o processamento vetorial e escalar, introduzindo sérios gargalos no sistema. Esta idéia está bem na linha das arquiteturas híbridas, unindo as melhores características dos modelos de fluxo de dados e von Neumann, mas deve haver ainda muito estudo nesta área para que ela torne-se viável.

Outra idéia que merece ser explorada é a utilização de instruções vetoriais, porém sem processadores vetoriais. A Unidade de Escalonamento de instrução sincronizaria apenas uma instrução, que seria dividida em várias outras quando chegasse na Unidade de Processamento. Nesse caso o comportamento da máquina seria bem mais previsível, mas também seriam necessárias alterações na arquitetura. Ambas as idéias de execução de instruções vetoriais merecem ser exploradas mais a fundo.

## 4.5 Resumo

A arquitetura da MX foi descrita neste capítulo. Inicialmente, foi analisada a arquitetura da MFDM, salientando seus pontos positivos e negativos. Estas informações, e mais as tendências atuais das pesquisas na área de arquiteturas híbridas serviram com base para o projeto da MX.

A MX é basicamente um multiprocessador de fluxo de dados com gerenciamento explícito de memória e capacidade para a execução seqüencial de blocos de instruções. Seus dois princípios básicos fornecem uma boa indicação de que a máquina possui uma relação custo/benefício melhor do que as disponíveis comercialmente no momento. O primeiro, de que qualquer processador pode executar

qualquer instrução habilitada, impede a ociosidade desnecessária de processadores. O segundo, de que a sincronização das instruções não ocorre no caminho crítico da execução, evita a ocorrência de bolhas, que comprometem o desempenho de muitas máquinas.

A arquitetura da MX é dividida em dois módulos principais, a Unidade de Escalonamento de Instruções (UEI) e a Unidade de Processamento de Instruções (UPI). A UEI realiza o escalonamento explícito das instruções à maneira de fluxo de dados. A UPI é responsável pelo processamento dos blocos de instruções, possuindo características comuns às arquiteturas von Neumann convencionais. O principal objetivo deste trabalho é garantir que o sistema de memória da UPI seja capaz de suprir as necessidades dos processadores independentemente de latência, validando, desse modo, o primeiro princípio do projeto.

# Capítulo 5

# Avaliação

No Capítulo 4 foi apresentada a arquitetura da MX, uma máquina de fluxo de dados que incorpora uma memória para armazenamento de dados, além de outras características do modelo von Neumann, para aumentar sua eficiência. Embora seja uma proposta preliminar e bastante abstrata, sendo portanto carente de muitos detalhes de implementação, faz-se necessária uma avaliação, não do seu desempenho, mas das reais condições de se tornar operacional. A situação ideal para avaliar a MX seria a construção de um simulador completo. Isto, no entanto, não foi possível, devido às sérias restrições de tempo impostas a este trabalho.

Primeiramente é analisada a funcionalidade do sistema de memória, ou seja, pretende-se garantir que a memória será capaz de alimentar os processadores com os dados necessários sem causar nenhuma degradação em desempenho. Então, um pequeno programa é apresentado como exemplo, para demonstrar que um compilador é capaz de gerar código de máquina executável sem muita dificuldade a partir de uma linguagem de programação de alto nível. Finalmente, são apresentados resultados de um trabalho de simulação não muito extenso, mas que vem endossar as idéias apresentadas sobre o sistema de memória e o funcionamento da MX como um todo.

## 5.1 Análise do Sistema de Memória

O sistema de memória é a parte mais crítica do atual estágio em que se encontra o projeto da MX. É dele que depende a garantia de funcionamento do primeiro princípio básico deste projeto, pelo qual qualquer processador pode executar qualquer instrução habilitada. Dessa forma pode ser eliminada a necessidade de balanceamento de carga, inevitável na maioria das propostas de arquiteturas

híbridas. Isto exige, porém, que a memória seja compartilhada por todos os processadores e que seja capaz de suprir as demandas por dados de modo a atingir o desempenho de 500 Mips esperado.

Devido à limitações tecnológicas, que impedem um único dispositivo físico de memória de obter um *bandwidth* tão elevado, será utilizado um esquema de memória entrelaçada. Desse modo, é possível obter um alto *bandwidth* através do processamento simultâneo de várias requisições em vários módulos de memória. Há alguns anos atrás havia muito ceticismo em relação ao desempenho da memória entrelaçada, com uma crença predominante de que M módulos de memória apresentariam um desempenho apenas proporcional à raiz quadrada de M. Atualmente sabe-se que com a utilização de filas nos módulos de memória e esquemas eficientes para a distribuição dos dados de modo a minimizar os conflitos nos acessos, esta questão pode ser solucionada. Além disso, no caso específico da MX, espera-se que o mesmo assincronismo inerente ao modelo de fluxo de dados que gera falta de localidade e impede a utilização eficiente de memória *cache*, proporcione um acesso bastante distribuído à memória.

### 5.1.1 Projeto da memória

Na apresentação do sistema de memória feita no Capítulo 4 não foi levada em consideração nenhuma informação a respeito da implementação da memória entrelaçada, como o *bandwidth* de memória esperado, o tempo de acesso dos dispositivos de memória e o número de módulos necessários. Além disso existem outras questões que afetam o desempenho de um memória entrelaçada e não foram consideradas. Agora, pretende-se demonstrar que o projeto físico da MX (e em especial, do sistema de memória) é perfeitamente viável, empregando-se componentes de uma tecnologia compatível com a dos atuais supercomputadores.

Em DFES [91] foi feita a opção por chips de memória utilizados em PCs, que são baratos e lentos, com o objetivo de construir um computador superescalar com o preço mais acessível ao usuário. No projeto da MX, a abordagem é um pouco diferente. Acredita-se que a visão de DFES é por demais acadêmica, e como todos os computadores de alto desempenho disponíveis atualmente utilizam tecnologia avançada, o importante é oferecer uma melhor relação custo/desempenho. Mais do que isto, a MX pretende oferecer um sistema mais fácil de utilizar, com uma linguagem de programação funcional na qual o paralelismo é implícito e que possibilite um processo de produção de software mais rápido.

Em primeiro lugar, deve-se determinar o *bandwidth* necessário ao sistema de memória, que pode ser calculado a partir do desempenho esperado da máquina. Este desempenho é da ordem de 500 Mips, o que significa que em média uma

instrução será executada a cada 2 ns. De uma maneira geral, para cada instrução são necessários três acessos à memória, dois para a leitura dos operandos e um para a escrita do resultado. Como as palavras de memória são formadas por 64 bits (8 bytes), um cálculo ingênuo produz como resultado um *bandwidth* de 12 Gb/s. No entanto, a MX possui a capacidade de executar blocos de instruções seqüencialmente, que se comunicam através de registradores e não precisam de acessos à memória. Estudos realizados em alguns grupos de pesquisa demonstram que o agrupamento médio de instruções de fluxo de dados se situa na faixa de 3 a 5 instruções [73, 46, 85]. Assumindo por segurança um mínimo de 3 instruções por bloco, as necessidades de acesso à memória ficam reduzidas a um terço do total inicial. Desse modo, o *bandwidth* máximo fica em 4 Gb/s, ou 500 milhões de palavras de 64 bits por segundo. Visto de outra forma, deve-se ter acesso a uma palavra a cada 2 ns, que coincide com a taxa de processamento das instruções. Este não é um valor alto, se comparado a outras arquiteturas de alto desempenho, como será visto na Seção 5.1.3.

O projeto físico da memória baseia-se em chips de memória RAM estática MOS com *tempo de acesso*[1] de 40 ns, que são utilizados no supercomputador NEC SX-2 [56]. Esta decisão foi tomada para simplificar a memória entrelaçada diminuindo o número de módulos, pois os chips de memória RAM dinâmica disponíveis comercialmente e utilizados nos PCs mais velozes são duas vezes mais lentos. Além disso, em memória RAM dinâmica o *tempo de ciclo*[2] corresponde a cerca de duas vezes o tempo de acesso, enquanto que em RAM estática estes tempos são em geral iguais [27].

Considerando que o acesso a uma palavra de 64 bits demore 40 ns, são necessários 40 ns / 2 ns = 20 módulos de memória entrelaçada para obter um *bandwidth* de 4 Gb/s. Este valor é por demais exato, sendo que para definir o número efetivo de módulos a ser utilizado no projeto deve-se levar em consideração dois fatores importantes:

(a) o *bandwidth* de 4 Gb/s foi calculado supondo-se um número médio de 3 instruções por bloco, mas para um número maior de instruções (possivelmente até 5) o *bandwidth* necessário diminui;

(b) por mais distribuídos que sejam os acessos aos módulos da memória, é praticamente impossível evitar que ocorram conflitos no acesso a um mesmo módulo, o que leva a um *bandwidth* efetivo menor que o calculado.

[1] O tempo de acesso da memória é definido como o tempo máximo necessário para ler uma palavra.

[2] O tempo de ciclo da memória é definido como o tempo mínimo que deve ser respeitado entre o início de duas operações sucessivas.

O primeiro fator tende a diminuir o número de módulos e o segundo a aumentá-lo, mas este último predomina, porque os acessos são assíncronos e imprevisíveis. Com base nisto, por segurança foi definido em 32 o número de módulos utilizados pelo sistema de memória da MX, com um *bandwidth* total de 6,4 Gb/s. Isto corresponde ao dobro do necessário (considerando 4 instruções por bloco) para obter um desempenho de 500 Mips operando em condições ideais.

Por outro lado, caso de confirmasse a hipótese de que o *bandwidth* efetivo de um sistema de memória entrelaçada seria proporcional somente à raiz quadrada do número de módulos utilizados, seriam necessários $20^2 = 400$ módulos de memória para atingir o *bandwidth* de 4 Gb/s estimado inicialmente. Apesar de alto, 400 não é um número assombroso (o NEC SX-2 utiliza 512 módulos de memória),mas a escalabilidade da máquina estaria fortemente comprometida.

Essa crença era sustentada pelo modelo de Hellermann, que determina o bloqueio de todos os acessos quando dois endereços fazem referência ao mesmo módulo de memória. Desse modo, este resultado não é inerente ao sistema de memória, nem às dependências de dados, mas deve-se simplesmente ao fato de não haver uma fila para armazenar as requisições que não puderam ser atendidas em um determinado ciclo de memória. Com a utilização de filas nos módulos, que é o caso da MX, pode-se obter um *bandwidth* linear, proporcional ao número total de módulos de memória [21, 17] e a hipótese a respeito da raiz quadrada pode ser posta de lado [64]. Considerando-se filas de tamanho infinito, o *bandwidth* efetivo aproxima-se do número de módulos de memória M. Como isto não é possível, admite-se suficiente a estimativa feita para a MX, considerando o dobro do *bandwidth* necessário.

No entanto, é importante lembrar que os estudos e simulações descritos em [21] têm como fundamento duas suposições básicas: não existem dependências de dados entre os acessos e estes têm um boa distribuição entre os módulos de memória. Em multiprocessadores convencionais von Neumann isto em geral não é tão fácil de obter, mas a MX é bastante beneficiada pelo modelo de fluxo de dados. Em primeiro lugar, o modelo de fluxo de dados é inerentemente paralelo e instruções com interdependências de dados nunca estão habilitadas a executar simultaneamente. Em segundo lugar, espera-se que a mesma característica assíncrona do modelo de fluxo de dados que causa a falta de localidade nos acessos à memória e impede a utilização de memória *cache*, produza um padrão de acessos razoavelmente bem distribuído.

A utilização de memória *cache*, aliás, não apresenta um desempenho muito bom no âmbito de aplicações intensivamente numéricas, embora para aplicações escalares consiga diminuir sensivelmente o tempo de acesso para a maioria dos dados. Aplicações numéricas utilizam freqüentemente vetores e matrizes muito

grandes, de modo que o acesso a um determinado elemento somente se repete após todos os outros elementos terem sido manipulados. Exceto no caso de problemas pequenos e inexpressivos, esses vetores e matrizes tendem a ser de tamanho consideravelmente maior do que qualquer *cache* real, fazendo com que as palavras sejam retiradas da memória *cache* antes de serem reutilizadas, resultando em uma baixa taxa de acertos.

Por este motivo, computações intensivamente numéricas são mais influenciadas pelo *bandwidth* da memória do que pelo tempo de acesso, e os supercomputadores evoluíram de maneira a ser relativamente insensíveis ao tempo de acesso da memória. Em processadores vetoriais, o tempo de acesso da memória contribui somente para uma penalização no tempo de preparação do vetor, mas não na taxa de execução das operações realizadas sobre ele.

Na MX também existe este tempo inicial de preparação sobre o qual incidem, além do tempo de acesso, os atrasos relativos às filas de entrada e saída , à rede de interconexão e ao controlador da memória. O acesso aos dados é feito através de um *pipeline*, que deve estar totalmente preenchido para que se obtenha o desempenho integral. Variações nas taxas de recebimento das requisições causam "bolhas" no *pipeline*, que certamente penalizam o desempenho. Felizmente o sistema de memória da MX (que pretende ser um multiprocessador de propósito geral, embora tenha inclinação ao processamento numérico) não faz distinção entre o acesso a dados estruturados e escalares. Sendo assim, pode atingir um bom desempenho tanto em processamento vetorial quanto escalar, o que não ocorre na maioria dos supercomputadores. A única ressalva é que, embora a MX consiga obter um alto *bandwidth* de memória, o tempo de acesso a uma posição específica é bastante elevado. Isto é devido não só ao longo caminho a percorrer, mas principalmente à existência das filas nos módulos de memória, que tendem a acumular várias requisições pendentes quando o sistema está funcionando a "pleno vapor" e causam atrasos adicionais.

Em arquiteturas de supercomputadores a intenção é sempre assegurar um *bandwidth* de memória alto e constante, mesmo que isto aumente o tempo de acesso. Entretanto, esquemas convencionais de armazenamento com distribuição seqüencial dos endereços de vetores pelos módulos de memória entrelaçada não oferecem essa garantia. A solução, então, é a distribuição mais ou menos aleatória dos endereços sobre os módulos de memória. Na Seção 5.1.4 são apresentados alguns esquemas eficientes de armazenamento para memória entrelaçada.

### 5.1.2 Balanceamento do *bandwidth*

Nas seções anteriores, buscou-se mostrar que o sistema de memória consegue obter um *bandwidth* suficiente para manter a máquina com um desempenho constante de 500 Mips. Entretanto, se houver gargalos em outras partes da máquina, grande parte desse desempenho pode ser desperdiçado. Para que isto não ocorra é necessário balancear o *bandwidth* global da máquina, ou seja, assegurar que todas as suas partes serão capazes de produzir um resultado a cada ciclo de 2 ns.

Os Elementos de Processamento (EPs) foram descritos como tendo uma capacidade total de processamento de 500 Mips. Para isto são utilizados 20 EPs, cada um composto de um processador RISC de 25 Mips e uma memória de instruções. Um processador com um *pipeline* de instruções operando a 25 Mips produz um resultado a cada 40 ns. Supondo que cada estágio do *pipeline* possua um ciclo fixo de 40 ns, existe uma equivalência direta com o tempo de acesso da RAM estática utilizada na memória de dados. Devido à necessidade de replicação (20 vezes) o custo de tal implementação pode tornar-se muito alto, recomendando a avaliação da possibilidade de serem utilizadas memórias RAM dinâmicas mais lentas e baratas. Nesse caso, com um tempo de acesso de 80 ns, o tempo de ciclo fica em torno de 160 ns. Para atingir o mesmo *bandwidth* será necessário buscar mais palavras de uma vez, o que pode ser obtido com o entrelaçamento da memória em um pequeno número de módulos. A decisão deve ser tomada em favor da opção que apresente a melhor relação custo/benefício. Em qualquer caso, a memória de instruções não representa nenhum empecilho ao projeto da MX.

Os Processadores na MX estão suscetíveis a um problema semelhante ao ocasionado pelas instruções de desvio nas arquiteturas RISC convencionais. O término da execução de um bloco de instruções somente pode ser detectado na fase de decodificação da instrução, e isto irá gerar uma bolha no *pipeline* interno do processador entre cada bloco, penalizando o desempenho. Entre as soluções possíveis, duas delas parecem mais viáveis:

(a) Codificar o número de instruções do bloco juntamente com a primeira instrução, de modo que o processador não precise esperar até a decodificação da última instrução para avançar para outro bloco. Assim, a bolha pode ser eliminada.

(b) Assumir a bolha, e aumentar o desempenho de pico para compensar a perda. Pode-se aumentar o número de processadores ou o desempenho individual de cada um. Com 4 instruções por bloco, o acréscimo terá que ser da ordem de 20 %.

Em princípio a primeira solução parece ser a melhor, mas ela requer um contador interno ao processador para a detecção do final do bloco.

Até agora o desempenho da máquina tem sido estabelecido somente em Mips, muito embora ela se destine também a aplicações numéricas. Em geral, existe uma diferença muito grande entre o desempenho em Mips e MFlops em computadores reais. Um fato importante na MX é a sua capacidade de fornecer dados de 64 bits suficientes para o processamento de 500 milhões de operações por segundo, sejam elas de ponto flutuante ou não. Para um desempenho de 500 MFlops são necessários 20 processadores de 25 MFlops, de acordo com as bases do projeto. A decisão de utilizar um processador mais potente (e conseqüentemente mais caro) para garantir 500 MFlops, ou um menos potente (talvez um processador de 25 Mips com coprocessador aritmético), e perder desempenho em MFlops, é algo que deve ser feito com base na viabilidade econômica da implementação. Ao nível em que o projeto está, pode-se garantir que isto é tecnicamente viável, pelo menos do ponto de vista do sistema de memória.

Na MFDM o principal gargalo é a Unidade de Agrupamento, ocasionado pela utilização de memória associativa. O desempenho de pico é ditado pela máxima taxa de emparelhamentos por segundo que pode ser alcançada, e não pode ser aproveitado integralmente pelo inevitável aparecimento das bolhas no *pipeline*. Na MX, o segundo princípio básico do projeto afirma que o agrupamento das instruções não ocorre no caminho crítico da execução, dessa forma evitando o aparecimento das bolhas que tanto penalizam o desempenho.

A aplicação deste princípio requer que a UEI seja capaz de fornecer um pacote executável à UPI a cada bloco de instruções executado. Isto vincula a velocidade necessária da UEI ao tamanho médio dos blocos de instrução. Para blocos de instrução com tamanho médio igual a 1, seria necessário um pacote a cada 2 ns. Para 2 instruções por bloco, 4 ns, e assim por diante. Como na MX somente instruções com dois operandos necessitam de sincronização e passam pela UEI (instruções com um operando são executadas dentro de um bloco na UPI), metade das fichas identificam o primeiro operando, e por isto não geram um pacote executável. Portanto, para que um pacote seja produzido a cada 2 ns, o ciclo interno da UEI precisa ser de 1 ns.

Uma vantagem da implementação da UEI é que ela pode ser replicada fisicamente, embora logicamente seja considerada uma só unidade. Além disso, internamente a cada unidade, pode-se utilizar memória entrelaçada para aumentar o desempenho. A Fila de Fichas e a Memória de Blocos não apresentam nenhum problema de implementação, a primeira por ser uma FIFO, e a segunda por conter informações somente de leitura. A memória de sincronização, entretanto, apresenta dois problemas adicionais. O processo de sincronização exige dois

ciclos para sua concretização, o primeiro para leitura e o segundo para escrita. Quando o primeiro operando chega, ele lê a posição indicando "vazio" altera para "cheio" e então o pacote é abortado. Quando o segundo operando chega, ela lê a posição indicando "cheio", altera novamente para "vazio" e o pacote é enviado à UPI. Portanto, a memória de sincronização exige o dobro do *bandwidth*. Além disso, a replicação tem que ser sustentada por uma função de distribuição pseudo-aleatória, de modo a não sobrecarregar demais certas unidades físicas, deixando outras ociosas.

A decisão a respeito do número de unidades replicadas e o nível de entrelaçamento de cada unidade só pode ser tomada com o auxílio de um trabalho extenso de simulação (a simulação da UEI a este nível não foi contemplada neste trabalho). Deve ser levada em consideração principalmente a relação custo/desempenho, pois dependendo dos resultados de simulação podem ser utilizadas menos unidades replicadas com memória RAM estática, ou então maior número de unidades com memória RAM dinâmica. Uma análise deste tipo somente pode ser levada a cabo com a definição precisa do custo de cada alternativa em contraste com o seu desempenho obtido por simulação.

Uma parte bastante crítica do sistema são os distribuidores e árbitros, que são encontrados em vários locais, como por exemplo, nos elementos de processamento e na entrada e saída da UEI. Eles representam um afunilamento no tráfego de informações e podem se transformar em sérios gargalos se não for tomado cuidado na implementação. Em termos de circuitos lógicos combinacionais um distribuidor pode ser considerado um demultiplexador e um árbitro, um multiplexador. Assumindo um atraso de 250 ps por porta, o tempo utilizado no NEC SX-2, pode-se utilizar até oito níveis de lógica para não ultrapassar os 2 ns requeridos, o que é bastante razoável. Não se pode prescindir da utilização de uma tecnologia avançada nessas regiões críticas, mesmo a um custo alto.

No presente projeto foi feita a opção por utilizar uma rede de interconexão entre as filas de entrada e saída e os módulos de memória baseada em múltiplos barramentos. Tal opção é bastante influenciada pelo projeto de DFES e entre as suas principais virtudes está o baixo custo que a implementação de múltiplos barramentos apresenta quando comparada a redes mais complexas. Resultados de simulação de DFES demonstraram que para uma configuração com 16 módulos de memória com tempo de acesso de 100 ns, filas com capacidade de 4 requisições em cada módulo e 4 barramentos com tempo de ciclo de 40 ns, pode-se obter um *bandwidth* de 512 Mb/s.

Para DFES esses resultados são perfeitamente justificáveis, mas estão aquém das necessidades da MX. Provavelmente, com a utilização de componentes de uma tecnologia mais avançada e o aumento do número de módulos de memória e

barramentos e da capacidade das filas, pode-se obter resultados muito melhores. Além disso o afrouxamento da condição de estrita seqüencialidade de DFES pode beneficiar a MX.

Uma estimativa do *bandwidth* esperado para a rede de interconexão poder ser realizada da seguinte maneira. Cada operação de leitura trafega duas vezes na rede de interconexão, de modo que cada bloco de instrução executado necessita de cinco acessos à ela. Considerando, como um exemplo, uma média de 3 instruções por bloco, para se manter dentro das especificações de desempenho um bloco deve ser processado a cada 6 ns. Isto então, gera um *bandwidth* aproximado de 6,7 Gb/s.

Nos últimos anos, o uso de redes de interconexão baseadas em múltiplos barramentos tiveram um uso crescente em projetos de multiprocessadores convencionais de memória compartilhada, com resultados encorajadores [16]. Uma decisão definitiva sobre a implementação somente pode ser conhecida através de simulações. Os resultados de simulação apresentados na Seção 5.3 apresentam uma forte indicação e que a abordagem com múltiplos barramentos é compatível com as necessidades da MX. Em todo caso, pode-se analisar o impacto da utilização de redes mais complexas neste projeto. Como exemplo, pode-se citar dois projetos da IBM que utilizam redes multiestágio com bastante sucesso. O RP3 [3] consegue um *bandwidth* da rede de interconexão de 13 Gb/s com um desempenho de 800 MFlops. O GF11 chega a 11 GFlops, com um *bandwidth* de 11,5 Gb/s. Este desempenho é seguramente mais do que suficiente para a MX, apresentando contudo as desvantagens de maior custo e tempos de latência mais elevados para o sistema de memória.

Além de provar a viabilidade técnica da máquina, o balanceamento do *bandwidth* de todos os seus componentes pode ser utilizado para prover uma outra característica importante, a escalabilidade. No projeto da MX foi tomado o cuidado de não esgotar toda a possibilidade de aumentar o seu desempenho quando necessário. Sem alterações drásticas na arquitetura, acredita-se que seja possível aumentar em até 4 vezes o desempenho, somente com a utilização de tecnologias mais avançadas. Os tempos de memória e portas lógicas considerados neste capítulo foram retirados do NEC SX-2, um projeto com quase 10 anos de existência. De lá para cá os componentes eletrônicos no mínimo dobraram de desempenho a um mesmo custo. Além disso, no projeto da memória de dados foram considerados apenas 32 módulos de memória entrelaçada, enquanto que o NEC SX-2 utiliza 512. Isto é um bom indício de que ainda há muito a explorar.

Outro ponto favorável à MX é a sua possibilidade de obter um desempenho efetivo próximo ao desempenho de pico, desde que haja suficiente paralelismo na aplicação. A MFDM, mesmo com todas as deficiências apresentadas, consegue

uma utilização dos processadores superior a 70% em uma configuração com 12 processadores [43, 42], enquanto processadores vetoriais em geral não passam de 50%. Como uma evolução da MFDM, espera-se um percentual maior para a MX.

### 5.1.3 Sistemas paralelos de memória

O *bandwidth* de 6,4 Gb/s projetado para a MX pode parecer um pouco elevado a uma primeira vista, embora tenha sido constatada a capacidade do sistema de memória em obtê-lo. Entretanto, muitas arquiteturas de alto desempenho conseguem obter um alto *bandwidth* de memória, utilizando memória entrelaçada e, em menor escala memória *cache*. Com o objetivo de validar os resultados da MX são apresentados agora alguns projetos de tais arquiteturas. Todos eles são processadores vetoriais, escolhidos por se encontrarem entre os mais velozes supercomputadores existentes atualmente e se basearem largamente na utilização de memória entrelaçada, como a MX.

O NEC SX-2 [56] consegue obter uma velocidade de pico de 1,3 GFlops e entre os supercomputadores aqui apresentados é o que possui maior *bandwidth* de memória. Com 512 módulos de memória entrelaçada e RAM estática com tempo de acesso de 40 ns, ele consegue ter acesso a 8 palavras de 64 bits a cada ciclo de máquina de 6 ns, perfazendo um *bandwidth* de 11 Gb/s.

O Cray X-MP [13, 47] em uma configuração com 4 processadores consegue chegar a uma velocidade de pico de 940 MFlops. Utiliza um nível de entrelaçamento de 64 módulos de memória e RAM estática com tempo de acesso de 34 ns, conseguindo endereçar até 8 palavras de 64 bits em um ciclo de máquina de 8,5 ns. Isto resulta em um *bandwidth* máximo de 7,5 Gb/s. O CRAY 2 e o CRAY Y-MP, com ciclos de máquina de 4,1 ns e 6 ns respectivamente alcançam um *bandwidth* maior, porém não foi possível ter acesso a uma documentação precisa a este respeito.

O CDC Cyber 205 [3, 47], ao contrário da maioria dos processadores vetoriais, tem uma organização memória-a-memória, em vez de registrador-a-registrador. O seu nível de entrelaçamento é menor, apenas 4 módulos, mas para compensar tem um barramento de dados de largura maior, podendo fazer referência a uma "super-palavra" de 512 bits (8 palavras de 64 bits) a cada ciclo de máquina de 20 ns. Com uma RAM estática de 80 ns, obtém um *bandwidth* de memória de 3,2 Gb/s. Seu desempenho de pico é de 400 MFlops, recorrendo para isso a 4 *pipelines* aritméticos de propósito geral.

O Fujitsu VP-200 [47] atinge 533 MFlops, e com 256 módulos de memória entrelaçada com tempo de acesso de 55 ns, pode fazer referência até 4 palavras de 64 bits a cada ciclo de máquina de 7,5 ns. Isto produz um *bandwidth* máximo de 4,2 Gb/s.

O IBM 3090 VF [3] atinge 828 MFlops e, podendo ter acesso a 3 palavras de 64 bits a cada ciclo de máquina de 14,5 ns, consegue obter um *bandwidth* de 1,6 Gb/s. Na realidade, ele não é um projeto integrado de um supercomputador, mas um processador vetorial[3] que pode ser conectado a um computador de grande porte IBM 3090. Desse modo, além da memória entrelaçada, pode utilizar a estrutura de memória *cache* do 3090 para suprir suas necessidades por acessos à memória.

A análise das informações resumidas sobre estas arquiteturas demonstra a viabilidade técnica em obter um *bandwidth* de 6,4 Gb/s para a MX. Mais do que isto, o fato de já terem transcorrido entre 5 e 10 anos desde a conclusão desses projetos pode ser visto como um ponto a favor do objetivo de escalabilidade. Não está sendo utilizada a tecnologia mais avançada do momento para não esgotar as possibilidades de aumento de desempenho e para que, se acaso simulações posteriores determinarem a existência de gargalos não identificados no projeto atual, não fique bloqueada a possibilidade de utilização de tecnologia mais avançada para resolver o problema.

### 5.1.4 Redução de conflitos em memória entrelaçada

Sistemas de memória entrelaçada, utilizados principalmente no âmbito de processamento vetorial, freqüentemente não conseguem manter um *bandwidth* alto e constante como seria desejável, devido aos conflitos que ocorrem no acesso aos módulos de memória. Conflitos podem ocorrer quando uma requisição é feita a um módulo ocupado ou quando requisições simultâneas são feitas a um mesmo módulo em um sistema que suporta múltiplos processadores. Entre os fatores que afetam a freqüência dos conflitos estão os seguintes: o número de módulos, a organização do sistema de memória e o *passo*[4]. O passo é a distância (em palavras de memória) entre elementos sucessivos de um vetor (ou matriz) numa seqüência de acesso.

Quando o passo e o número de módulos são primos, praticamente todos os acessos são livres de conflitos. Uma dificuldade em utilizar um número de módulos que não seja uma potência de dois é a necessidade de realizar uma divisão inteira no hardware de endereçamento. O caso mais crítico ocorre quando o passo é um múltiplo do número de módulos, pois então todas as referências à memória são dirigidas ao mesmo módulo e o *bandwidth* da memória é reduzido à freqüência de acesso a um único módulo.

Na descrição do sistema de memória no Capítulo 4, supõe-se um armazenamento linear simétrico, ou seja, em um sistema com M módulos de memória, um

[3] Vector Facility - VF

[4] stride

determinado endereço E será sempre armazenado no módulo $E$ mod $M$. Considerando uma matriz de dimensões $M \times M$, onde o elemento $A_{i,j}$ é armazenado no módulo $j$, é possível fazer acesso simultâneo a todos os elementos de uma linha, desde que eles residam em módulos diferentes. O acesso a colunas (com passo M), entretanto, causará grande penalização em desempenho, porque todos os elementos estarão em um mesmo módulo. Adotando uma estratégia de armazenamento diferente, onde o elemento $A_{i,j}$ é armazenado no módulo $(i + j)$ mod $M$, todos os elementos em qualquer linha ou coluna da matriz[5]. A estarão contidos em módulos de memória distintos. Tal esquema de armazenamento é chamado *assimétrico* [6] [75], e, em particular neste exemplo, ele é periódico. Essencialmente, um esquema assimétrico é um mapa de armazenamento para todos os endereços no espaço de endereçamento.

A implementação de um esquema assimétrico pode ser feita por software ou hardware. Por software, o endereço efetivo deve ser calculado através de uma função (no exemplo, $(i + j)$ mod $M$), e é considerado pura sobrecarga[7] para o processador. A implementação por hardware é mais aconselhável em sistemas que são mais fortemente influenciados por *bandwidth* do que pelo tempo de acesso da memória. Neste grupo estão os processadores vetoriais e a MX. Nesse caso, um nível a mais de circuitos é utilizado para implementar a função de armazenamento. Em geral são necessárias $O(MlogM)$ portas lógicas para este fim. É proposto um esquema de armazenamento assimétrico baseado em lógica XOR chamado *irregular*[8] é proposto em [54], onde para implementar o hardware de endereçamento são utilizadas somente $logM$ portas. Neste esquema, a rede de interconexão tem um papel fundamental no processo de decodificação dos endereços.

Uma proposta mais recente [74] argumenta ser necessária também a sincronização dos processadores, além da distribuição ótima dos dados pelos módulos de memória para aumentar o *bandwidth* efetivo de memória. Aparentemente, devido aos processadores aguardarem o retorno de suas requisições à memória, podem ocorrer problemas de contenção que anulam os benefícios de qualquer esquema de distribuição de endereços. Nesse sentido, a MX leva uma grande vantagem, uma vez que após retornar da memória, os dados podem ser repassados a qualquer Elemento de Processamento desocupado.

Outros esquemas têm sido propostos para resolver o problema causado pelo passo. Um sistema não periódico proposto em [88], aumenta o desempenho do

[5] Nesse caso, as linha são deslocadas à direita em um módulo, relativo à linha superior
[6] skewed
[7] overhead
[8] scrambled

sistema de memória para vários passos considerados, e sua implementação em hardware causa um atraso de apenas dois ciclos de máquina no tempo de acesso da memória. A técnica de expansão de dimensões [22] é realizada em tempo de compilação, mas tem o inconveniente de consumir memória adicional.

Além dos conflitos causados pelo passo, que ocorrem em mais freqüência em computadores SIMD, devem ser levados em consideração aqueles causados por requisições simultâneas a um mesmo módulo de memória, no âmbito de um computador MIMD como a MX. Foi visto que, com a existência de filas, pode-se obter um *bandwidth* proporcional ao número de módulos de memória utilizados. No entanto, estas conclusões são obtidas com base em algumas simplificações que são feitas no padrão de acessos à memória de cada processador. Em [49] são considerados com maior profundidade os efeitos de referências escalares e vetoriais no desempenho de um sistema de memória entrelaçada. É proposto um sistema de entrelaçamento variável, que divide dinamicamente o número de módulos e o espaço de endereçamento entre os processadores para evitar conflitos, mas pode obter todo o benefício do número integral de módulos quando for necessário. Entrelaçamento variável é também utilizado na arquitetura TERA [4] e em Monsoon [59].

Em Monsoon, o entrelaçamento variável é utilizado com base em estruturas de dados, onde um conjunto de módulos que armazena uma estrutura é chamado de subdomínio. Cada estrutura tem um padrão de acessos diferente em relação aos módulos de memória. No entanto, o critério utilizado para criar subdomínios não é bem especificado.

Na MX, é difícil prever a ocorrência de conflitos nos acessos aos módulos de memória, devido ao assincronismo próprio de uma arquitetura de fluxo de dados. Existe uma forte crença de que o desempenho não seja fortemente afetado, mas isto somente pode ser determinado por simulações. Por isto, talvez alguns dos esquemas apresentados tenham que ser mais cuidadosamente considerados. O problema de passo não deve ter um efeito negativo tão grande quanto nos processadores vetoriais, porque os elementos de um vetor não necessariamente sofrem acessos em seqüência. Na realidade, a ordem dos acessos é indeterminada. O entrelaçamento variável pode ser utilizado na implementação do gerente da memória. Certos segmentos cujos dados estão sendo muito utilizados em um determinado momento podem ser atribuídos a subdomínios diferentes, de modo a minimizar conflitos.

## 5.2 Programa exemplo

Um programa para a MX é semelhante aos executados em várias arquiteturas híbridas, onde a comunicação entre instruções num mesmo bloco[9] é feita por registradores e em blocos distintos através da memória. Fichas não carregam dados, apenas sinais de controle. Nesta seção é apresentado um programa exemplo para a MX para demonstrar que o processo computacional é bem definido para ela, e que é possível construir um compilador para ela a partir de uma linguagem de alto nível. O mesmo programa é apresentado para a MFDM, juntamente com o programa fonte escrito na linguagem SISAL [30], ambos retirados de [42].

A Figura 5.1 mostra um programa para integração numérica na linguagem SISAL, que calcula a área sob a curva $y = x^2$ entre $x = 0,0$ e $x = 1,0$, utilizando aproximação por trapézios com $x$ sendo incrementado com o intervalo constante de $0,02$. Na Figura 5.2 está o mesmo programa em forma de grafo de fluxo de dados para a MFDM. O grafo é um exemplo de código reentrante, isto é, que reutiliza parte de si mesmo. O modelo de fluxo de dados dinâmico adotado na MFDM possui um mecanismo para a criação de múltiplas instâncias de um sub-grafo. No caso de um laço, os dados identificam a instância a que pertencem através de um campo denominado *nível de iterações*. Este campo é incrementado, a cada ciclo, implicitamente alocando novos recursos de memória na unidade de emparelhamento. É interessante notar, nesse exemplo, que instruções que possuem um único operando constituem mais de $62,5\%$ do total. Mesmo sem necessidade de sincronização, elas consomem um ciclo inteiro de anel para sua execução.

A Figura 5.3 mostra o mesmo programa para a MX. Na Figura 5.3a está o código do programa em forma de linguagem de máquina convencional. A representação em grafo de fluxo de dados não é muito adequada para a MX, por não fazer distinção entre fluxo de dados e de controle. As instruções CGR, BRR, ADR e MLR são adaptações do conjunto de instruções da MFDM. As demais foram descritas na Seção 4.4.1. Os indicadores de bloco, apresentados na Figura 5.3b, devem ser gerados juntamente com o código, sendo necessário um indicador para cada bloco de instruções. O indicador do bloco 1 realiza apenas uma sincronização, enquanto que o do bloco 2 busca os operandos na memória sem necessitar de sincronização. O bloco 1 não é realmente necessário, apenas evita a necessidade de uma sincronização no bloco 2 a cada iteração. A Figura 5.3c apresenta um diagrama funcional, uma representação gráfica dos blocos para que se possa visualizar o seu relacionamento, e os locais onde ocorre sincronização.

[9] thread

```
function Integrate (returns real)

for initial
    int := 0.0;
    y   := 0.0;
    x   := 0.02;
while
    x < 1.0
repeat
    int := 0.01 * (oldy + y);
    y   := oldx * oldx;
    x   := oldx + 0.02;
returns
    value of sum int

end function
```

Figura 5.1: Programa para integração numérica na linguagem SISAL

Uma comparação superficial entre os programas das Figuras 5.2 e 5.3 poderia levar a conclusões incorretas. O programa para a MX necessita de mais instruções e reduz o paralelismo, em relação à MFDM. Em primeiro lugar, o paralelismo no programa da MFDM não é muito grande, no máximo 3, e não desempenha um papel muito importante no tempo total de execução. Em segundo lugar, apesar de o número de instruções ser maior na MX, elas são executadas seqüencialmente dentro dos blocos. Na MFDM, cada aresta significa um ciclo de anel, que é aproximadamente uma ordem de grandeza maior que a do ciclo do processador.

Na MFDM, um ciclo de anel é determinado pelo tempo que um pacote leva para percorrer todos os estágios do *pipeline*. Na MX, embora não exista propriamente um anel, esta medida pode ser considerada como a duração do intervalo entre o término da execução de uma instrução que produz uma ficha e a chegada de uma pacote executável habilitado por esta ficha. São considerados os atrasos produzidos por todos os estágios da máquina no trajeto, excluindo a possibilidade de conflitos nos acessos à memória (que são imprevisíveis). Uma suposição aceitável é que este ciclo de anel na MX corresponda a dez vezes o ciclo de processador de 40 ns, que produz o valor de 400 ns. Supõe-se também por simplicidade que uma instrução será executada a cada ciclo de processador (isto na realidade

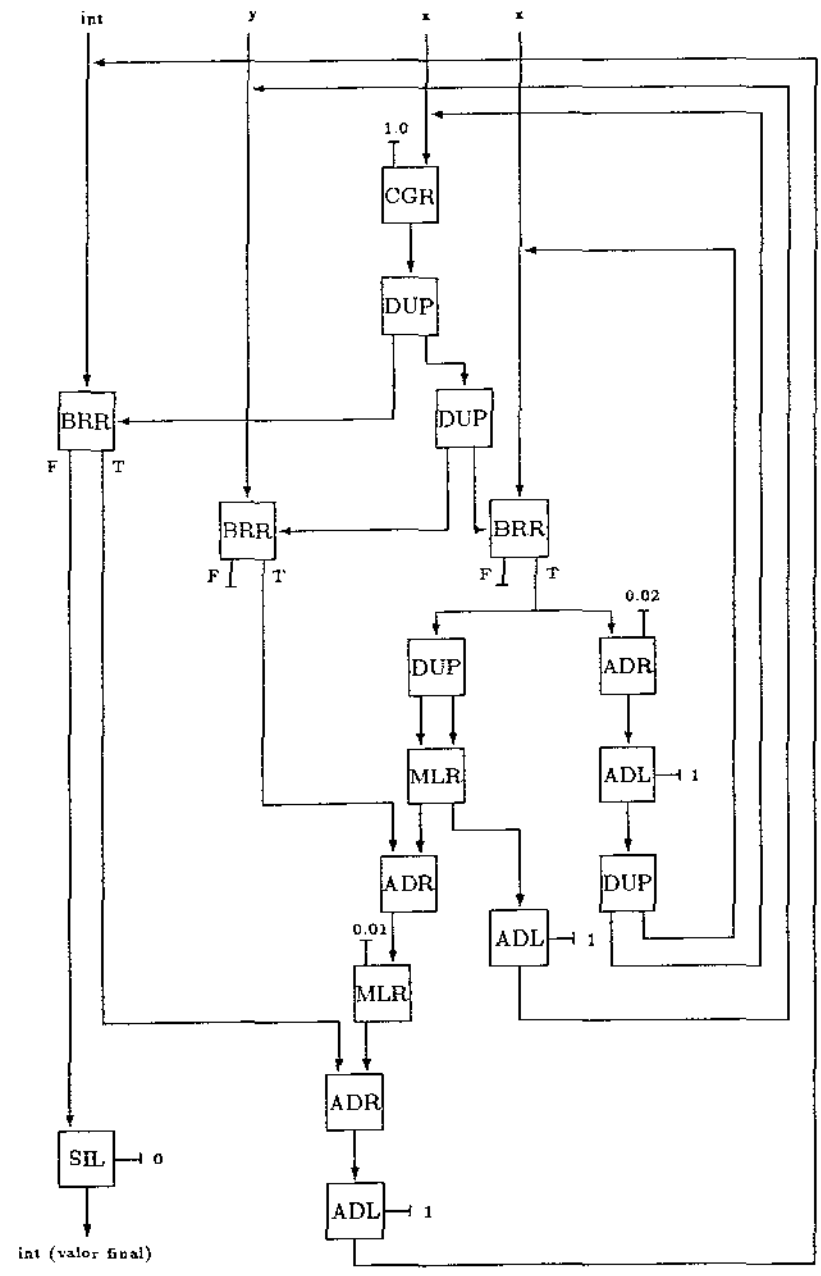


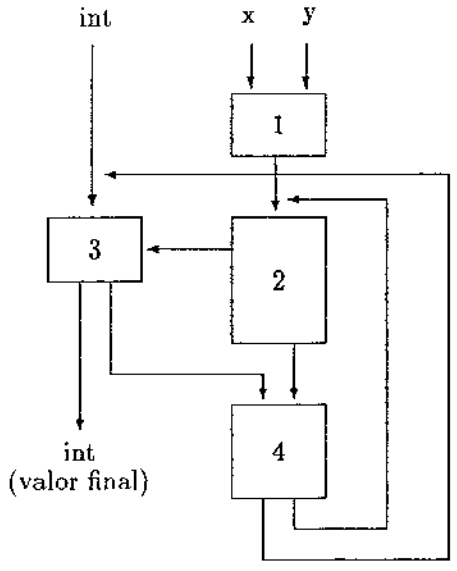

Figura 5.2: Programa para integração numérica na MFDM

```
BLOCO 1

1: SGN 2

BLOCO 2

2: CGR 1.0, x, Rres
3: STR Rres, res, -
4: SGN 14
5: BRR Rres, 6, -
6: ADR x, 0.02, Rx
7: MLR x, x, Ry
8: ADR y, Ry, Raux
9: MLR Raux, 0.01, Raux
10: STR Rx, x, -
11: STR Ry, y, -
12: STR Raux, aux, -
13: SGN 17

BLOCO 3

14: BRR res, 15, 16
15: SGN 17
16: SGN endereço final

BLOCO 4

17: ADR int, aux, Rint
18: STR Rint, int, -
19: SGN 14
20: SGN 2
```

(a)

```
BLOCO 1

1: -, -, sinc1

BLOCO 2

2: x, y, -

BLOCO 3

14: res, -, sinc2

BLOCO 4

17: int, aux, sinc3
```

(b)




(c)

Figura 5.3: Programa para integração numérica na MX: (a) código; (b) indicadores de bloco; (c) diagrama de blocos

somente ocorre quando o *pipeline* do processador está cheio) e que uma aresta (fluxo de controle) é executada a cada ciclo de anel. Levando em consideração o número de instruções e arestas no caminho crítico, a execução do programa da Figura 5.2 consome 7 ciclos de processador e 7 ciclos de anel por iteração, o que na MX corresponde a 3,08 $\mu$s. O programa da Figura 5.3 consome 16 ciclos de processador (a primeira instrução não é considerada pois somente é executada na primeira iteração), mas apenas 2 ciclos de anel, correspondendo a um tempo de execução de 1,44 $\mu$s. Isto significa que, para este exemplo, a execução de instruções em bloco é mais que 2 vezes mais rápida que a execução do grafo de fluxo de dados correspondente. Mesmo reduzindo o paralelismo e aumentando o número de instruções, para este caso pode-se ver que a abordagem de codificação de programas utilizada na MX é mais eficiente que a tradicional. Isto comprova as deficiências atribuídas aos grafos de fluxo de dados na Seção 3.2.

Outra informação importante que pode ser retirada deste exemplo é a respeito da utilização de memória. No programa para a MFDM, cada ciclo do laço necessita de um novo contexto, indicado pelo nível de iteração, que implicitamente aloca recursos de memória. No caso da MX, a quantidade de memória necessária é consideravelmente menor, devido à eficiência dos mecanismos von Neumann utilizados. Logicamente isto somente é possível no caso de laços com corpo pequeno e com dependências de dados entre as iterações que é o caso deste exemplo. Para laços do tipo "para todos"[10] ou com maior paralelismo e assincronismo interno a cada iteração, não convém limitar o paralelismo desta maneira. Nesses casos cada iteração necessitaria de um contexto e a comunicação entre elas seria idêntica à chamada de procedimento (Seção 4.4.2), com toda a sobrecarga adicional que isto implica. O que está claro, é que em várias situações podem ser utilizadas técnicas von Neumann com resultados muito favoráveis.

## 5.3 Simulação

Um estudo de simulação é uma forma reconhecida para estimar-se o desempenho de uma nova máquina. Com este objetivo em mente, foi desenvolvido este estudo, no qual se buscou comprovar a eficácia das principais características do projeto da MX. O desejo maior foi a validação da capacidade do sistema de memória em manter os Elementos de Processamento (EPs) ocupados. Embora fosse possível simular somente o comportamento do sistema de memória, foi feita a opção pela simulação da máquina como um todo, o que possibilitou a obtenção de informações bem mais abrangentes. O modelo, os algoritmos e os resultados

[10] forall

da simulação são apresentados a seguir.

### 5.3.1 Modelo e algoritmo de simulação

O modelo de simulação da MX é baseado em atividades e pode ser visto na Figura 5.4. O modelo pode ser descrito em termos de 3 conceitos básicos: atividades, servidores e filas. As atividades ocupam os servidores por determinado período de tempo e comunicam-se através das filas. À medida que o tempo simulado avança, as condições para o início ou o fim de cada atividade são examinadas.

A execução de instruções não é simulada diretamente. Em vez disso, simula-se o fluxo de fichas (ou pacotes) pelo sistema, que podem ter significados diferentes em cada componente. O término da simulação é atingido quando determinada quantidade de fichas tiver circulado pelo modelo. Os resultados de simulação foram obtidos com o modelo já em um estado estável, sendo para isso realizada um pré-simulação com uma amostragem de 500 fichas. O paralelismo é mantido constante durante todo o tempo de duração de uma simulação. Isto significa que cada bloco de instruções gera apenas uma nova ficha e não existe sincronização na UEI. Os resultados obtidos são basicamente estatísticas sobre a utilização de servidores e filas.

Os componentes da MX representados com atividades e servidores no modelo são:

- *Unidade de Escalonamento de Instruções (UEI):* Recebe informações da fila de saída e produz resultados para a fila de entrada. Internamente é representada como uma caixa preta. Isto já é suficiente para a obtenção de informações sobre a demanda necessária para se adequar a certas configurações da UPI e características de programas.

- *Fila de Entrada (FE):* É responsável por fornecer os pacotes executáveis aos EPs. A cada pacote (ficha) recebido são emitidas duas requisições de leitura para a memória com endereços gerados aleatoriamente. As unidades internas (servidores) da FE ficam ocupadas até que os resultados dessas requisições sejam recebidos.

- *Elementos de Processamento (EPs):* Representam o processamento dos blocos de instrução. O número médio de instruções por bloco ($\mu$IB) é informado como parâmetro de simulação.

- *Fila de Saída (FS):* Envia resultados recebidos dos EPs à memória e fichas à UEI. Estas duas funções são desvinculadas, ou seja, as fichas podem ser

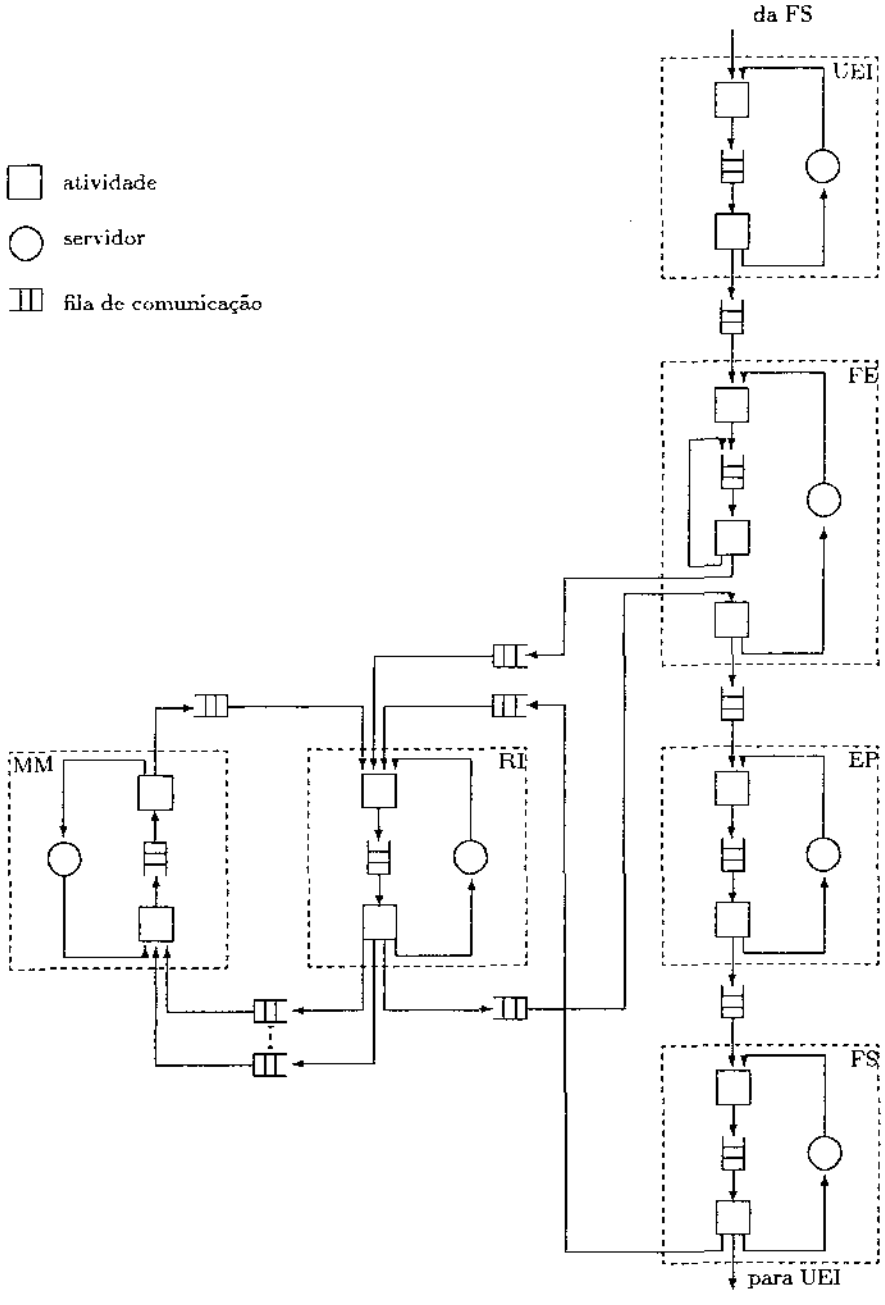



Figura 5.4: O modelo de simulação da MX

| *nome* | *quantidade* | *tempo (ns)* |
|---|---|---|
| EP | 20 | 40 |
| MM | 32 | 40 |
| RI | 20 | 20 |
| UEI | 1 | 1 a 14 |
| FS | 4 | 20 |
| FE | 32 | 20 |

Tabela 5.1: Parâmetros de simulação

enviadas à UEI sem depender do envio de resultados à memória. Como na FE, existe a geração aleatória de endereços de memória.

- *Rede de Interconexão (RI):* Forma a interface externa do sistema de memória. Monitora os Módulos de Memória (MM), a FE e a FS atribuindo prioridades a elas nesta ordem. As requisições de leitura/escrita de memória são inseridas nas respectivas filas internas dos MM de acordo com o endereço gerado.

- *Módulos de Memória (MM):* São representados pelos 32 módulos estabelecidos no projeto com suas respectivas filas. Requisições a módulos (servidores) ocupados aguardam a sua liberação. Requisições de leitura geram resultados a FE por intermédio de RI.

A estrutura interna da UEI, o Gerente da Memória e o Sistema de Comunicação não foram modelados, visto não serem imprescindíveis para a obtenção de resultados significativos para o estágio atual deste projeto.

### 5.3.2 Resultados de simulação

Desde o início das simulações, a característica observada que mais influênciou o desempenho do sistema de memória foi a média de instruções por bloco ($\mu$IB). Ela foi a principal fonte de orientação para a definição de uma configuração ideal para a MX, como o número de unidades internas da FE, FS e RI e a demanda exigida da UEI.

A Tabela 5.2 apresenta valores percentuais de utilização dos servidores da UPI em função de $\mu$IB. Os parâmetros de configuração da máquina utilizados nestas simulações podem ser vistos na Tabela 5.1. Alguns deles são especificações de

| $\mu$IB | %EP | %MM | %RI | %FE | %FS |
|---|---|---|---|---|---|
| 1 | 36,91 | 69,22 | 92,19 | 100,00 | 92,19 |
| 2 | 75,44 | 69,26 | 92,24 | 100,00 | 92,31 |
| 3 | 100,00 | 62,06 | 82,99 | 78,74 | 81,84 |
| 4 | 100,00 | 46,52 | 62,21 | 50,16 | 61,05 |
| 5 | 100,00 | 37,19 | 49,76 | 37,67 | 48,76 |
| 6 | 100,00 | 30,99 | 51,86 | 29,89 | 40,60 |
| 7 | 100,00 | 26,57 | 44,44 | 25,12 | 34,89 |

Tabela 5.2: Utilização percentual em função de $\mu$IB

projeto (MM e EP); os demais foram obtidos em simulações preliminares. Além dos parâmetros mostrados na Tabela 5.1, as simulações foram realizadas com paralelismo médio igual a 100, 1000 amostras (cada amostra representa a execução de um bloco de instruções) e 8 filas internas para cada Módulo de Memória.

Pode-se ver na Tabela 5.2 que a partir de $\mu$IB = 3 os EPs já apresentam uma taxa de utilização de 100%. Na realidade, valores de $\mu$IB inferiores a 3 não são relevantes para este estudo, uma vez que o agrupamento médio de instruções em fluxo de dados se situa na faixa de 3 a 5 (v. Seção 5.1). Além do aproveitamento integral dos EPs, para $\mu$IB = 3 há um balanceamento ideal entre os componentes da máquina. Não há gargalos nem ociosidade. Para $\mu$IB inferior a 3, a FE torna-se um sério gargalo, assim como para $\mu$IB acima de 5 a maioria dos componentes apresenta ociosidade. Embora seja uma característica dos programas executados, $\mu$IB tem um papel fundamental na arquitetura da MX.

A uma primeira vista o sistema de memória parece estar operando com bastante folga, com MM = 62,06 % de ocupação para $\mu$IB = 3 e um máximo de 69,22% para $\mu$IB = 1. Isto é, no entanto, devido aos conflitos nos acessos aos módulos de memória, que deixam certos módulos ociosos enquanto requisições esperam nas filas de outros módulos ocupados. O projeto do sistema de memória, com 32 módulos, levou esta questão em consideração, sendo as taxas de ocupação compatíveis com as necessidades dos EPs.

Simulações adicionais para $\mu$IB = 2 com valores aumentados para alguns parâmetros conseguiram obter 100% de utilização dos EPs e 92% de MM. O *bandwidth* máximo de memória para esta configuração é de 6,4 Gb/s e o requerido para os EPs não apresentarem ociosidade com $\mu$IB = 2 é 6,0 Gb/s. A utilização de 92% é compatível com estes valores, e demonstra que o desempenho efetivo

do sistema de memória pode se aproximar bastante do ideal teórico.

Outras simulações, com $\mu$IB = 1 apresentaram 100% de utilização para MM. Embora fosse esperado um alto desempenho da memória devido às filas existentes em cada módulo, estes resultados foram uma agradável surpresa. Eles foram obtidos com uma configuração superdimensionada para a MX e paralelismo superior àquele encontrado na maioria das aplicações, mas conseguiram demonstrar o potencial do mecanismo utilizado. Neste último caso, foram observados tempos médios de espera nas filas de 1 $\mu$s, com até 60 requisições pendentes em um módulo. Estes dados podem ser contrastados com os obtidos em condições normais (Tabela 5.2) onde os tempos de espera situam-se na faixa de 10 a 30 ns e um máximo de 7 requisições pendentes.

Na Seção 5.1.1 foi questionado o desempenho de uma rede de interconexão baseada em múltiplos barramentos. Os resultados comprovam as estimativas de cerca de 6,7 Gb/s para o *bandwidth* quando $\mu$IB = 3. Supôs-se um tempo de 20 ns (50 MHz) para os barramentos, e, para uma quantidade de 20 barramentos o desempenho foi de acordo com o esperado. Outras simulações foram realizadas com 30 ns (33 MHz aproximadamente, freqüência já utilizada hoje em dia em microcomputadores) sendo observados resultados semelhantes.

As filas de entrada e saída puderam ter suas unidades internas quantificadas com este trabalho de simulação. Sem a simulação não seria trivial obter uma estimativa confiável da demanda sobre esses componentes. No Capítulo 4 a Fila de Saída já havia sido descrita com tendo uma estrutura simples, o que se confirmou nas simulações. Com apenas 4 unidades internas ela é capaz de cumprir adequadamente sua função. Já a Fila de Entrada necessita de 32 unidades internas para obter desempenho semelhante, considerando $\mu$IB $\geq$ 3. Para valores de $\mu$IB inferiores a 3 ela torna-se o gargalo do sistema, com 100% de utilização. Essa relação de 1 para 8 entre o número de unidades internas dos dois componentes pode ser compreendida levando em consideração que ao tempo de atraso específico da Fila de Entrada devem ser acrescidos os atrasos relativos ao sistema de memória (MM e RI) para se quantificar o tempo de ocupação de cada unidade interna no processamento dos pacotes.

O paralelismo médio ($\mu$Par) utilizado como parâmetro para a maioria das simulações foi da ordem de 100, uma vez que a MX destina-se a aplicações altamente paralelas. É interessante, porém, avaliar também o desempenho da máquina executando aplicações com paralelismos diferentes. Na Tabela 5.3 pode-se observar que a partir de $\mu$Par = 50 já se obtém taxas de utilização altíssimas para os EPs, chegando a 100% com $\mu$IB = 5. Para $\mu$Par = 30 ocorre subutilização dos EPs e de MM, não tendo sido registrados resultados ótimos com $\mu$IB $<$ 10. Por outro lado, uma vez alcançada a utilização integral dos EPs, o paralelismo

| $\mu$PAR | $\mu$IB = 3 | | $\mu$IB = 4 | | $\mu$IB = 5 | |
|---|---|---|---|---|---|---|
| | %EP | %MM | %EP | %MM | %EP | %MM |
| 30 | 65,90 | 40,54 | 76,42 | 35,06 | 83,71 | 30,68 |
| 50 | 95,60 | 58,70 | 99,91 | 45,76 | 100,00 | 36,65 |
| 100 | 100,00 | 62,06 | 100,00 | 46,52 | 100,00 | 37,19 |
| 200 | 100,00 | 62,06 | 100,00 | 46,52 | 100,00 | 37,19 |

Tabela 5.3: Influência do paralelismo

| UEIt | %EP | %UEI |
|---|---|---|
| 2 | 100,00 | 32,75 |
| 4 | 100,00 | 67,03 |
| 6 | 100,00 | 100,00 |
| 8 | 76,34 | 100,00 |
| 10 | 61,17 | 100,00 |

Tabela 5.4: Ciclo interno da UEI

excedente não beneficia a máquina[11]. Isto pode ser observado nas taxas idênticas de utilização para $\mu$Par = 100 e $\mu$Par = 200. A obtenção de 100% de utilização dos EPs com $\mu$Par = 50 é uma informação importante para a MX. Como exemplo, Monsoon em uma configuração com 20 processadores necessitaria de $\mu$Par = 160 para chegar a este patamar.

O trabalho de simulação realizado concentrou-se na UPI, principalmente na influência dos sistema de memória sobre o aproveitamento da capacidade de processamento dos EPs. A UEI não foi modelada internamente, sendo considerada uma caixa preta da qual somente interessa saber neste momento o desempenho necessário para que a UPI não fique ociosa. Na Seção 5.1.1 foi apresentada uma estimativa do ciclo interno da UEI baseada em $\mu$IB. Esta previsão foi inteiramente confirmada, sendo utilizados tempos de produção de fichas de 2 a 14 ns para $\mu$IB variando de 1 a 7 nas Tabelas 5.2, 5.3 e 5.5 (ou seja, cada instrução adicional em $\mu$IB proporciona 2 ns para a UEI). A Tabela 5.4 mostra que para $\mu$IB = 3 o ciclo interno ideal da UEI é 6 ns. Ciclos mais longos deixam os EPs

[11] Por sinal, pode ser até prejudicial (vide Seção 6.1.2)

| $\mu$IB | $\mu$LSM | tINI |
|---|---|---|
| 3 | 210 | 344 |
| 4 | 189 | 392 |
| 5 | 180 | 460 |

Tabela 5.5: Atrasos do sistema de memória

ociosos, penalizando o desempenho da máquina. Ciclos mais curtos não oferecem nenhuma contribuição e deixam a UEI ociosa, além de onerar sua implementação. Resultados semelhantes foram obtidos para outros valores de $\mu$IB.

O sistema de memória projetado consegue obter um bom desempenho, fornecendo dados a uma taxa compatível com a capacidade de processamento dos EPs. Os resultados apresentados acima foram obtidos, no entanto, com $\mu$Par constante durante as simulações. Em aplicações com paralelismo muito variável no tempo, certamente haverá penalizações devido à latência do sistema de memória, como já havia sido previsto na Seção 5.1.1. O sistema de memória funciona como um *pipeline*, de modo que eventuais "bolhas" não podem ser eliminadas. Devido a esses fatores, deve-se também analisar o desempenho do sistema de memória sem considerá-lo já num estado estável.

Na Tabela 5.5 são apresentadas duas importantes informações a este respeito, em função de $\mu$IB. A primeira é a latência média do sistema de memória ($\mu$LSM), que é a duração do intervalo entre a chegada de um pacote à UPI e o momento em que ele fica disponível para os EPs. Aí estão considerados os tempos individuais de cada componente, mais o tempo gasto em ciclos de espera entre os componentes e nas filas de memória. Pode-se ver que em situações de baixo paralelismo o desempenho do sistema de memória fica totalmente comprometido, para valores menores de $\mu$IB.

A segunda informação é o tempo inicial dispendido para os EPs alcançarem um regime de 100% de utilização (tIni). Para os valores de $\mu$IB apresentados na Tabela 5.5 este tempo gira em torno de 400 ns. Isto corresponde ao tempo em que a MX executa 200 instruções em condições ideais. Sob condições de paralelismo excessivamente inconstante é bastante provável que muitos ciclos dos EPs sejam gastos esperando o sistema de memória se recompor.

### 5.3.3 Observações

As seguintes observações foram feitas com base nos resultados obtidos:

- *Configuração para a MX:* A definição de uma configuração ideal para a MX deve levar em consideração vários fatores, entre eles $\mu$IB. Cruzando os resultados de simulação obtidos com os de pesquisas a respeito de agrupamento de instruções em fluxo de dados, chega-se ao valor de $\mu$IB = 3, para que se tenha segurança. Isto tem uma influência direta no ciclo de produção de fichas da UEI, que fica fixado em 6 ns. Esta informação é muito importante para um projeto mais detalhado da UEI. Os outros parâmetros podem ser tomados como aqueles da Tabela 5.1, fixando em 8 o número de *buffers* de cada módulo de memória.

- *Importância de $\mu$IB:* A média de instruções por bloco ($\mu$IB) é um dos fatores de maior impacto no desempenho do sistema de memória, como pode ser observado no decorrer desta seção. Isto deve-se ao fato de ela determinar a quantidade de acessos à memória dos programas em execução, pois quanto maior for $\mu$IB, menor a demanda de memória exigida. O sistema de memória teve sua configuração determinada em projeto (Seção 5.1) com o objetivo de manter os EPs ocupados permanentemente, mantendo um desempenho constante de 500 Mips. O objetivo deste estudo de simulação foi dar um embasamento real às afirmativas teóricas. Desse modo, a discussão concentrou-se não em alterar a configuração do sistema de memória, mas em encontrar meios para satisfazer da melhor maneira as condições estabelecidas.

  A importância de $\mu$IB torna-se, então, bastante evidente. Com valores altos de $\mu$IB pode-se obter um desempenho ainda superior a 500 Mips com a mesma configuração do sistema de memória. Técnicas de agrupamento de instruções em fluxo de dados estão sendo estudadas [84] e devem ter um grande impacto na características de programas para a MX. Na Seção 6.1.1 são apresentadas sugestões de técnicas de compilação visando aumentar o desempenho da MX, o que pode levar, inclusive, a repensar-se a arquitetura da máquina, visando a execução de blocos de instruções maiores.

- *Utilização dos EPs:* Foi visto que se pode obter 100% de utilização dos EPs em condições normais a partir de $\mu$IB = 3 e em condições favorecidas a partir de $\mu$IB = 2. Este fato, no entanto, não significa que a MX apresenta um ganho[12] linear em função do número de EPs. Na realidade, ele indica que a arquitetura da MX possui um grande potencial para a utilização eficaz da sua capacidade de processamento, o que não ocorre atualmente na maioria

[12] speed-up

dos supercomputadores. Existem algumas condições que favoreceram esses resultados e que certamente têm influência sobre o desempenho da máquina:

- Não foram simulados programas reais, apenas o fluxo de fichas.
- O paralelismo foi mantido constante. Não houve produção de fichas (*fork*) nos EPs, nem sincronização (*join*) na UEI.
- A estrutura interna da UEI não foi modelada.
- O gerente da memória não foi modelado.
- A estrutura interna dos EPs não foi modelada.

Isto, porém, não desmerece os resultados obtidos, pois a cada etapa de projeto deve-se utilizar as ferramentas que tragam os resultados necessários. Nesse sentido, a simulação realizada cumpriu o seu papel.

- *Utilização dos MMs:* Como já havia sido previsto, pôde-se comprovar que o modelo de Hellermann está incorreto. Ele diz que um sistema de memória com M módulos consegue obter apenas o desempenho equivalente à raiz quadrada de M. Nas simulações realizadas os MMs atenderam às expectativas para todos os valores de $\mu$IB considerados, mesmo que para isto fosse necessário superdimensionar a configuração da máquina. A utilização de filas nos módulos de memória foi decisiva para a obtenção deste resultado.

## 5.4 Resumo

A avaliação da MX feita neste capítulo baseou-se em três aspectos:

(a) Uma análise do sistema de memória, a fim de apresentar argumentos suficientes para comprovar a sua capacidade de fornecer dados a uma taxa que permite desempenho real próximo aos 500 Mips esperados. Com uma tecnologia de componentes utilizada em projetos com 5 a 10 anos de existência, o sistema de memória da MX consegue obter um *bandwidth* de 6,4 Gb/s. Um ponto fundamental para chegar a este valor é o abandono da antiga crença, de que para M módulos de memória entrelaçada, o desempenho seria equivalente à raiz quadrada de M.

(b) A apresentação de um programa exemplo, para confirmar que, a partir de uma linguagem de alto nível é possível gerar um programa em linguagem de máquina para a MX. Foi visto também, que o modelo de execução MX, baseado em blocos de instruções, é superior aos grafos de fluxo de dados.

(c) Um estudo de simulação no qual ficaram comprovadas as suposições teóricas a respeito do desempenho do sistema de memória e da MX como um todo. Foi visto que a média de instruções por bloco ($\mu$IB) é um dos fatores de maior influência neste desempenho e que, a partir de $\mu IB = 3$, consegue-se obter 100% de utilização dos Elementos de Processamento (EPs).

# Capítulo 6

# Conclusão

Este trabalho abordou as principais questões relacionadas à introdução de mecanismos von Neumann tradicionais em arquiteturas de fluxo de dados. Seu principal produto foi a arquitetura MX, apresentada no Capítulo 4 e avaliada no Capítulo 5. Conclui-se que uma arquitetura baseada nos princípios de projeto MX consegue obter um desempenho médio próximo ao desempenho de pico em condições reais. Resultados de simulação também indicam a adequação dos componentes internos da máquina aos princípios estabelecidos no Capítulo 4.

## 6.1 Trabalhos futuros

Um projeto completo exige muito mais do que as idéias aqui apresentadas. Seriam necessários vários estudos semelhantes a este para esclarecer alguns pontos obscuros que foram somente alinhavados. Apenas o sistema de memória recebeu maior atenção, por se tratar do objetivo primordial deste trabalho. Alguns temas para pesquisa, identificados ao longo da realização deste trabalho, são agora apresentados como sugestões para trabalhos futuros. Entre essas, algumas são apenas aprimoramentos e otimizações; outras, no entanto, constituem necessidades essenciais para o funcionamento e utilização da máquina.

### 6.1.1 Linguagens de Programação

Idealmente, vários paradigmas de programação deveriam ser suportados pela MX, desde linguagens puramente funcionais até linguagens tradicionais com semântica imperativa. Embora estas últimas não sejam muito adequadas à programação de máquina paralelas, sua larga utilização em sistemas computacionais de alto

desempenho torna impraticável o desperdício de tanto software até hoje já produzido. No entanto, numa fase inicial os esforços poderiam concentrar-se no suporte a uma única linguagem de programação —SISAL [30]—, na qual foi escrito o programa exemplo da Seção 5.2. SISAL é uma linguagem de atribuição única, não puramente funcional que pode ser utilizada em máquinas convencionais (multiprocessadoras ou não), mas é particularmente adequada a máquinas não convencionais, como a MFDM. Sua semântica é determinística, apresentando um enorme potencial de paralelismo. Programas em SISAL são compilados para uma linguagem intermediária gráfica e independente de máquina chamada IF1, e então convertidos para o código de máquina desejado. Para execução na MFDM o código IF1 é transformado em grafos de fluxo de dados por um sistema de compilação composto de três fases, sendo dependentes de máquinas os formatos intermediários subseqüentes.

Para a implementação da linguagem SISAL na MX deve-se partir do formato IF1 e aproveitar todo o conhecimento gerado na construção do sistema de compilação para a MFDM. Dois aspectos importantes da arquitetura MX devem ser analisados com mais cuidado no processo de compilação: a separação entre os fluxos de dados e de controle e a execução seqüencial de blocos de instruções. Na MFDM, as fichas são responsáveis pela transmissão de dados e de controle, mas na MX esta restrição não é necessária. A transmissão de controle é ainda realizada pelo envio de fichas de habilitação, mas a transmissão de dados é efetuada diretamente através da memória. Para isto, os endereços de memória relativos à base do segmento onde os dados serão armazenados deverão ser determinados durante a fase de compilação, enquanto as fichas indicativas da presença dos operandos nesses endereços serão produzidas durante a execução do programa.

Os blocos de instruções MX correspondem às linhas de controle [1] da maioria das arquiteturas híbridas. Dentro deles a transmissão de controle é seqüencial e os dados são passados através de registradores. Embora possam ocorrer blocos de instruções com apenas uma instrução, isto somente faz sentido para instruções que geram fichas ou requisições. Como conseqüência a maioria dos blocos tem no mínimo duas instruções. A identificação dos blocos de instruções num programa é uma tarefa básica do compilador. No Grupo de Fluxo de Dados da Unicamp estão sendo desenvolvidos trabalhos com o objetivo de analisar grafos de fluxo de dados e identificar grupos de instruções cuja execução seqüencial pode ser interessante [85, 55]. Embora essa análise esteja apoiada na MFDM, muitos resultados são independentes de máquina e bastante animadores. Espera-se que os algoritmos desenvolvidos nesses trabalhos possam ser utilizados, com as necessárias modi-

[1] threads

ficações, no ambiente MX. Além disso, deverão ser estudadas outras propostas de geração de código para arquiteturas híbridas que executam blocos de instruções seqüenciais [24, 48, 73, 81].

A arquitetura MX, na sua concepção atual, não permite que uma instrução tenha mais do que dois operandos, uma característica herdada da MFDM e fundamentada na simplicidade de projeto. Embora esta restrição não possua um efeito muito negativo para instruções isoladas, quando estendida para blocos seqüenciais ela pode tornar-se bastante indesejável. Com a limitação de duas entradas por bloco, o tamanho médio dos blocos diminui, enquanto a necessidade de acessos à memória e sincronizações aumenta. Isto pode prejudicar o desempenho e dificultar a codificação de regiões críticas para o sistema operacional, que devem ser executadas atomicamente. Uma solução completa e genérica somente pode ser obtida através de modificações na arquitetura. No entanto, na maior parte dos casos, pode-se conseguir uma solução paliativa que dobra o número de entradas.

A palavra de dados da MX possui 64 bits, mas pode-se fazer referência a palavras de 32 bits. Assim, em blocos que somente utilizam dados de no máximo 32 bits, pode-se obter 4 entradas, posicionando cada dupla em uma palavra de 64 bits. Embora voltada ao processamento científico, onde é muito comum a utilização de operações de ponto flutuante de precisão dupla, boa parte das instruções de um programa e praticamente a totalidade das de um sistema operacional não utilizam tais recursos. Mesmo essa solução sendo limitada, o impacto na geração dos blocos é grande. Como um exemplo, no programa da Figura 5.3 poderiam ser eliminados os blocos 3 e 4 com a adoção desta técnica, cuja implementação requer algumas pequenas modificações na arquitetura, para efetuar a sincronização dos operandos. Os indicadores de bloco devem ser estendidos com um campo contendo o número de fichas a sincronizar e a memória de sincronização deve realizar a função de contador e comparador. Um pacote só poderá ser habilitado quando o valor na memória de sincronização for igual ao do indicador de bloco.

Com relação à geração de código, duas importantes fontes de otimização devem ser consideradas:

(a) *Eliminação de sincronização redundante*: esta otimização permite a eliminação de arestas de um grafo quando puder ser provado que os dados já estão disponíveis na memória. Na Figura 6.1 pode-se ver dois exemplos de eliminação de sincronização redundante. Entre os vértices $v_i$ e $v_j$ não são necessárias duas arestas de ligação, uma vez que na MX as fichas que fluem através das arestas não carregam dados, mas apenas sinais de sincronização. Do mesmo modo, não é necessária a aresta entre os vértices

$v_l$ e $v_n$, pois existe uma ligação indireta através de $v_m$ e pode-se garantir que os resultados de $v_l$ estarão disponíveis na memória quando $v_n$ for escalonado, economizando-se sincronizações redundantes. As arestas eliminadas estão tracejadas na Figura 6.1 mostrando que com este método economizam-se duas sincronizações redundantes, uma conseqüência importante da separação dos fluxos de dado e de controle.

(b) *Redução da ocupação de memória*: uma maneira trivial de associar posições de memória a dados é simplesmente atribuir a cada operando de cada instrução um determinado deslocamento dentro de um segmento. Para minimizar o tamanho do segmento poder-se-ia reutilizar uma posição várias vezes durante a execução de um bloco de código. Isto é, seria desejável associar a mesma posição de memória a diferentes instruções ao mesmo tempo em que se preserva o determinismo dos resultados para todas as possíveis ordens de execução. Para que duas instruções, $v_i$ e $v_j$, possam compartilhar a mesma posição de memória, deve-se garantir que sua execução simultânea é impossível. Existem duas maneiras de garantir tal exclusão mútua:

- Existe uma dependência dupla entre $v_i$ e $v_j$, isto é, ambas as entradas de $v_j$ são dependentes, através de algum caminho, das saídas de $v_i$.
- As instruções $v_i$ e $v_j$ aparecem em lados opostos de um comando condicional.

Na Figura 6.1 podem ser vistos alguns casos de dependência dupla, como entre $v_i$ e $v_j$, $v_j$ e $v_m$, e $vl$ e $v_n$, onde os operandos dessas instruções podem ocupar as mesmas posições dentro do segmento. Na Figura 5.3 as variáveis utilizam as mesmas posições de memória para as várias iterações. Embora conceitualmente a cada iteração sejam criadas novas variáveis que passam resultados intermediários entre si, a implementação pode realizar uma otimização, utilizando a mesma posição de memória de maneira transparente.

É importante notar que os vértices da Figura 6.1 foram considerados instruções simples, para facilitar o raciocínio. Na realidade, na maior parte dos casos eles seriam blocos de instruções.

### 6.1.2 Controle de paralelismo

O modelo de fluxo de dados consegue expor muito paralelismo, mesmo em programas que não utilizam algoritmos paralelos especiais. No entanto, esse sucesso "excessivo" pode ser prejudicial. À medida em que mais paralelismo é exposto,

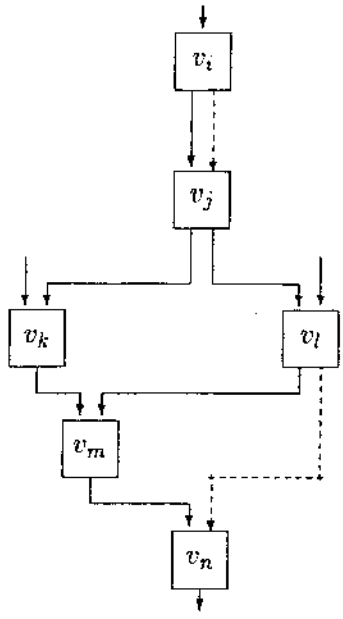



Figura 6.1: Otimizações em um grafo de fluxo de dados

mais recursos são necessários. A menos que sejam tomadas precauções adequadas, programas com grau de paralelismo elevado podem saturar, e até provocar *deadlock*, mesmo numa máquina de tamanho bastante razoável.

O excesso de paralelismo surge principalmente da execução de laços e de funções recursivas. As máquinas baseadas no modelo de fluxo de dados dinâmico exploram paralelismo entre as iterações de um laço *desdobrando-o* [2]. Nesta técnica as iterações de um laço são ativadas tão cedo quanto possível — o que pode levar a uma simultaneidade quase total — e operações nas várias iterações executam quando seus dados de entrada estão presentes. Por outro lado, funções recursivas geram árvores de ativação de tamanho imprevisível. Em uma arquitetura como a MX, onde cada ativação de um bloco de código identifica um novo contexto e recebe um segmento para sua execução, está claro que a demanda excessiva por recursos de memória pode tornar-se crítica para o processamento. É necessário, então, controlar o paralelismo a fim de limitar a utilização de recursos.

O controle de paralelismo pode ser feito através de métodos de hardware ou software. O principal exemplo de um método de hardware está na MFDM, onde uma unidade especial chamada *throttle* [67] foi acoplada ao sistema para controlar a liberação de nomes de ativação. Quando a máquina atinge um determinado

[2] loop unfolding

nível de utilização dos recursos, o *throttle* começa a retardar a alocação de contextos, isto é, a concessão de nomes de ativação. Este método apresentou bons resultados na MFDM.

Uma possível solução por software é limitar o desdobramento de laços, parametrizando durante a compilação o número de iterações que podem executar concorrentemente. Esta é a idéia por trás da técnica de *limitação de laços*[3] [23], que supõe a alocação de um conjunto limitado de registros de ativação a iterações sucessivas de um laço. Desse modo, uma janela deslizante com um número fixo de iterações pode estar executando em paralelo durante a execução do laço, limitando consideravelmente os recursos utilizados. O número de iterações ativas simultaneamente é determinado em tempo de execução com base na configuração e carga da máquina.

Outra maneira de controlar por software o paralelismo em um laço é permitir a execução sobreposta de diversos estágios de cada iteração, através de uma técnica desenvolvida para as arquiteturas VLIW [3] chamada *software pipelining*. Uma proposta de implementação de *software pipelining* para Monsoon que utiliza somente um registro de ativação por laço é apresentada em [15]. Na MFDA, onde a execução de laços é realizada por um método utilizado nas arquiteturas que seguem o modelo de fluxo de dados estático, a questão da demanda excessiva de recursos não chega a causar problemas [35]. Tal método, baseado em arestas de reconhecimento[4], pode ser visto com um caso onde o *software pipelining* é de granularidade fina.

Uma decisão sobre um controle de paralelismo eficiente para MX deverá exigir uma análise detalhada dos métodos apresentados e de outros que porventura existirem. Aqui serão apresentadas apenas algumas sugestões que podem ser úteis. O método por hardware utilizado na MFDM é sem dúvida o mais genérico porque trata qualquer excesso de paralelismo, independente da situação na qual ele foi gerado. Ao gerente da memória poderia ser adicionada a capacidade de controle de paralelismo a custo de hardware adicional, em uma espécie de união das funções da Sub-unidade de Alocação da Memória de Estruturas e do *throttle* da MFDM. No entanto, perder-se-ia a capacidade de reaproveitamento de contextos sem a necessidade de liberar e alocar novamente, como ocorre na técnica de limitação de laço. Na MFDM, alocar um contexto significa designar um nome de ativação que implicitamente identifica recursos de memória na Unidade de Agrupamento. Já na MX esta atividade é mais complexa, compreendendo a procura explícita de um segmento de memória de tamanho adequado ao bloco de

[3] loop bounding
[4] acknowledgement

código invocado. Seria extremamente ineficiente alocar e liberar vários segmentos idênticos durante a execução de um laço. Portanto, para a execução de laços a técnica de limitação de laço mostra-se mais eficiente que o *throttle*. Sua maior deficiência é não ser adequada ao tratamento de funções recursivas, mas o fato de aplicações científicas basearem-se principalmente na exploração de paralelismo entre iterações de laços pode ser um ponto a seu favor.

A técnica de *software pipelining*, embora eficiente para controlar paralelismo, parece ser restritiva demais, ou seja, pode expor menos paralelismo do que a máquina é capaz de explorar. No entanto, para laços com dependências entre iterações (e que por este motivo exibem menor paralelismo) a sua relação custo/benefício é muito boa. Somente um segmento é necessário e a restrição ao paralelismo não chega a causar problemas.

A imagem que se tem após esta análise preliminar, é que os três métodos apresentados possuem prós e contras e a união deles seria a melhor solução: limitação de laço para laços sem dependências entre iterações; *software pipelining* para laços com dependências entre iterações, e um mecanismo de hardware para controlar o paralelismo em excesso que porventura ainda venha a ocorrer. Isto, no entanto, em geral não é aplicável. Propostas que procuram reunir muitas características consideradas importantes geralmente tornam-se ineficientes.

### 6.1.3 Estruturas de dados

O tratamento de estruturas de dados possui alguns inconvenientes em máquinas de fluxo de dados. No modelo de fluxo de dados puro, dados não são armazenados, ocasionando um fraco desempenho para aplicações científicas, que utilizam muitos vetores e matrizes. Algumas propostas têm sido feitas para solucionar este problema [36], entre as quais destaca-se a que se baseia em Estruturas-I [12], principalmente no contexto de aplicações científicas. Estruturas-I são estruturas não estritas[5], apropriadas para o tratamento de vetores, que utilizam bits de presença para indicar se um elemento já foi produzido e está pronto para ser consumido. A memória de estruturas da MFDM [71] possui suporte para a utilização de Estruturas-I, embora dê preferência à utilização de estruturas estritas.

No Grupo de Fluxo de Dados da Unicamp foi desenvolvido um trabalho com o objetivo de reavaliar o armazenamento de estruturas de dados na MFDM e propor alterações [69]. Embora algumas boas idéias tenham sido apresentadas, elas não são adequadas para a MX. Na MX, tanto dados estruturados quanto

[5]Uma estrutura não estrita fica disponível aos processos consumidores mesmo antes da definição de todos os seus elementos. Em oposição, uma estrutura estrita requer a presença de todos os seus elementos antes que possa ser usada.

escalares são armazenados explicitamente, o que representa uma grande evolução em relação ao modelo de fluxo de dados puro, e necessita de soluções próprias. Como visto na Seção 4.3.3, na MX as estruturas de dados têm um tratamento idêntico a segmentos, e podem ser sincronizantes ou não-sincronizantes. Estruturas sincronizantes são estritas e necessitam de uma sincronização global, que é realizada pelo gerente da memória. No atual estágio do projeto, somente isto foi definido a respeito de estruturas de dados. Alguns tópicos importantes a serem estudados são:

- *Estruturas não estritas.* Sua implementação depende de alterações na arquitetura, embora sua necessidade ainda precise ser demonstrada.
- *Armazenamento eficiente de estruturas de dados.* Foi discutido na Seção 5.1.4.
- *Sincronização global para estruturas estritas.* Deve ser determinada a maneira como este procedimento será realizado. Em [69] são apresentadas algumas sugestões.
- *Alocação de estruturas de dados em segmentos de dados.* Estruturas cuja utilização restringe-se somente ao procedimento que as criou podem ser alocadas juntamente com os dados escalares no segmento, para reduzir o número de requisições ao gerente de memória em tempo de execução.
- *Recuperação de espaço.* Segmentos de dados escalares são liberados após o término do procedimento ativado. Estruturas de dados necessitam de um mecanismo diferente para a recuperação de espaço de memória não mais utilizado.
- *Reutilização de estruturas de dados.* A semântica funcional exige que uma nova estrutura seja criada a cada alteração, o que gera uma alta sobrecarga. No entanto, em muitos casos pode-se reaproveitar as estruturas sem perder a funcionalidade do modelo. A técnica de limitação de laço permite a reutilização de estruturas internas ao laço [23].

### 6.1.4 Sistema operacional

Para tornar a MX efetivamente utilizável, um grande passo será provê-la de um sistema operacional. Sistemas operacionais para máquinas paralelas possuem um nível maior de complexidade: sendo eles próprios programas paralelos, necessitam de mecanismos para controlar o acesso concorrente a certas estruturas de dados

que mantêm informações sobre o gerenciamento de recursos da máquina [3]. Entre os sistemas operacionais para máquinas paralelas von Neumann, os que mais se destacam são algumas variações do UNIX [68].

Para a MX, porém, em princípio não será necessário um sistema operacional multiusuário, que introduziria um nível maior de complexidade. Pode-se argumentar nesse sentido que aplicações que demandam alto desempenho não devem sofrer influência de outras aplicações. Um sistema operacional monousuário não requer mudanças na arquitetura (que não possui suporte para memória virtual, por exemplo) e pode ser perfeitamente capaz de administrar os recursos da máquina, levando em consideração o seu inerente paralelismo interno.

A implementação de sistemas operacionais em máquinas de fluxo de dados esbarra geralmente na criação de regiões críticas, que são essenciais para o gerenciamento de recursos. Isto ocorre porque o modelo de fluxo de dados é livre de efeitos colaterais, não sendo possível evitar a execução de determinada instrução que esteja habilitada. Entre os métodos propostos para implementar o gerenciamento de recursos em máquinas de fluxo de dados encontram-se os seguintes:

- Uma função de emparelhamento capaz de diferenciar as fichas destinadas ao gerente manipulando o nível de iteração do rótulo [8].
- Uma ficha especial chamada gatilho[6] controla o acesso de uma única requisição ao gerente, sendo as demais recirculadas, em um processo de "espera ativa"[7] [20].
- Uma Estrutura-I é utilizada como uma fila de entrada para serializar o acesso das requisições ao gerente, permitindo preservar a consistência em estruturas de dados essenciais [12].

Na MX, a possibilidade de executar instruções em blocos e a existência de uma memória para o armazenamento de resultados intermediários facilitam bastante o processo de gerenciamento de recursos. Em máquinas com memória distribuída, como Monsoon e EM-4, o benefício é ainda maior, pois uma vez que um bloco de instruções inicia a execução ninguém mais tem acesso aos dados, criando implicitamente uma região crítica. A memória compartilhada da MX é condição essencial, de modo que é necessário adaptar algum dos métodos das arquiteturas de fluxo de dados puras para a criação de regiões críticas. Dos métodos apresentados o mais conveniente é o terceiro, mas a MX não possui suporte para Estruturas-I. Este é um dos motivos que levaram à proposta da utilização de

[6] trigger
[7] busy-waiting

um computador hospedeiro para a realização das tarefas de gerenciamento de recursos na primeira etapa do projeto.

### 6.1.5 Projeto detalhado

De modo geral, e com exceção do sistema de memória o projeto da MX foi apresentado em um nível bastante abstrato. Mesmo assim, ainda há muito a ser definido. Alguns pontos importantes são:

- *Detalhamento do conjunto de instruções*, bem como dos procedimentos a serem tomados em algumas condições especiais, como o tratamento de exceções.
- *Projeto da arquitetura interna do processador.* Processadores comerciais não podem ser utilizados devido às instruções especiais incorporadas à MX. Neste item está incluído o relacionamento do processador com a memória de instruções.
- *Definição das funções do gerente da memória.* Durante este trabalho várias idéias propostas envolveram o gerente da memória. Deve ser tomado o cuidado de não torná-lo um gargalo para a máquina.
- *Projeto físico da máquina*: quando a arquitetura estiver completamente detalhada deverá ser feito o projeto físico com definição dos circuitos necessários e a tecnologia mais adequada à implementação.

## 6.2 Comentários finais

Este trabalho destinava-se originalmente a projetar uma memória de armazenamento de dados escalares para a MFDM. O objetivo era provar que a introdução de um mecanismo von Neumann tradicional, como o gerenciamento explícito de memória, poderia contribuir para a eliminação de algumas deficiências atribuídas às arquiteturas de fluxo de dados. Entretanto, logo ficou óbvia a inadequação da MFDM para este fim. Com o baixo desempenho (da ordem de 2 Mips) obtido pelo protótipo, apenas uma pequena quantidade de módulos de memória entrelaçada seria suficiente para atingir o *bandwidth* requerido. Esta informação se confirma mesmo utilizando-se memórias RAM dinâmicas com tempos de acesso compatíveis com a época e considerando-se a antiga e restritiva hipótese de que o desempenho do sistema seria proporcional à raiz quadrada do número de módulos de memória. Restaram então, duas possíveis alternativas:

(a) alterar a tecnologia dos componentes do protótipo por outra mais veloz,

(b) projetar uma nova arquitetura de alto desempenho baseada na MFDM, mas superando as suas maiores deficiências, de modo a oferecer uma plataforma conveniente para os testes do sistema de memória.

Foi escolhida a segunda alternativa, devido à generalidade e flexibilidade que daria ao projeto.

A MX surgiu, então, da observação de que algumas das maiores deficiências apresentadas pela MFDM e outras arquiteturas de fluxo de dados possuem soluções eficientes nas arquiteturas von Neumann, que não devem ser esquecidas em novos projetos. Essa possibilidade é hoje contemplada por vários projetos de pesquisa internacionais, cujos frutos estão sendo as arquiteturas híbridas. Este trabalho, em particular, apresenta duas contribuições positivas neste sentido. Em primeiro lugar, o projeto da arquitetura da MX, um multiprocessador de fluxo de dados que incorpora características híbridas. Embora ainda em nível abstrato, ele apresenta princípios de funcionamento bem definidos e representa o ponto de partida para uma série de trabalhos de pesquisa que são necessários para complementá-lo. Em segundo lugar, e como principal contribuição, apresenta fortes argumentos favorecendo a utilização do modelo de memória compartilhada em um multiprocessador de fluxo de dados. Foi demonstrado que o sistema de memória da MX é capaz de obter um *bandwidth* compatível com as exigências dos processadores, independente dos conflitos nos acessos aos módulos de memória entrelaçada. Mais importante do que isto, este *bandwidth* pode ser aumentado conforme houver a necessidade de uma maior velocidade de processamento. Neste caso, a proposta MX difere da maioria das arquiteturas híbridas, que utilizam memória distribuída para a simplificação do projeto, mas apresentam graves problemas com respeito à rede de interconexão e balanceamento de carga.

A MX representa uma boa opção para uma máquina paralela de alto desempenho, justamente pela preocupação intensa que houve no seu projeto no sentido de eliminar as deficiências mais graves das arquiteturas de fluxo de dados e von Neumann. No entanto, ela própria apresenta algumas limitações já identificadas, relacionadas a seguir:

- As instruções (blocos de instruções) podem ter no máximo 2 operandos, uma característica comum a várias arquiteturas de fluxo de dados. Na MX, porém, o número de operandos tem um efeito direto sobre o tamanho dos blocos, que deve ser o maior possível. Nos casos em que não se manipulam dados de 64 bits , o número de operandos pode ser dobrado, mas uma solução genérica exige mudanças na arquitetura.

- O sistema de memória impõe um longo tempo de acesso a dados individuais. Isto é um reflexo da utilização de um esquema de memória entrelaçada voltado ao oferecimento de alto desempenho sob intensa utilização, com um caminho muito grande a ser percorrido pelos dados desde a requisição até o consumo. Uma de suas conseqüências é um fraco desempenho do sistema de memória sob baixo paralelismo.

- Instruções vetoriais não são suportadas diretamente pela arquitetura (Seção 4.4.3), o que seria bastante conveniente numa máquina com inclinação para aplicações científicas.

- A arquitetura não oferece suporte para memória virtual, o que impede sua utilização compartilhada. Esta é, entretanto, uma característica questionável numa arquitetura de alto desempenho.

Várias críticas têm sido dirigidas tanto às arquiteturas von Neumann quanto às arquiteturas de fluxo de dados. Ao longo deste trabalho, a intenção foi sempre salientar que a união das melhores características de ambos os modelos pode resultar em uma arquitetura híbrida capaz de explorar o paralelismo de maneira natural e eficiente. Uma prova disto são os benefícios que podem ser observados com a introdução, em uma arquitetura de fluxo de dados, de um mecanismo para gerenciamento explícito da memória e execução seqüencial de instruções.

É indispensável a continuidade das pesquisas na área de arquiteturas híbridas. Talvez já estejamos perto de encontrar uma solução final adequada à exploração do paralelismo e a MX seja um bom caminho a seguir. Talvez uma solução mais conclusiva somente seja encontrada quando surgirem alguns conceitos ainda não desenvolvidos. Independente disto, a mais importante conclusão a que se chega é: não se deve desprezar as lições duramente aprendidas no passado. Os extremos não costumam ser a melhor solução.

# Bibliografia


[1] ACKERMAN, W. B. Data flow languages. *IEEE Computer*, v. 15, n. 2, p. 15–25, Feb. 1982.

[2] AGARWAL, A. et al. APRIL: a processor architecture for multiprocessing. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 17. *Proceedings...* May 1990. p. 104–114.

[3] ALMASI, G. S.; GOTTLIEB, A. *Highly parallel computing.* Benjamin/Cummings, 1989.

[4] ALVERSON, R. The Tera computer system. In: INTERNATIONAL SYMPOSIUM ON SUPERCOMPUTING, 1990. *Proceedings...* June 1990. p. 1–6.

[5] ARVIND; CULLER, D. E. Dataflow architectures. *Annual Reviews in Computer Science*, v. 1, p. 225–253, 1986.

[6] _____. _____. In: THAKKAR, S. S. (ed.). *Selected reprints on dataflow and reduction architectures.* IEEE Press, 1987. p. 79–101.

[7] ARVIND; EKANADHAM, K. Future scientific programming on parallel machines. *Journal of Parallel and Distributed Computing*, v. 5, n. 5, p. 460–493, Oct. 1988.

[8] ARVIND; GOSTELOW, K. P.; PLOUFFE, W. *An asynchronous programming language and computing machine.* Relatório técnico, Department of Information and Computer Science, University of California, Irvine, Sept. 1978.

[9] ARVIND; HELLER, S. K.; NIKHIL, R. S. Programming generality and parallel computers. In: INTERNATIONAL SYMPOSIUM ON BIOLOGICAL AND ARTIFICIAL INTELLIGENCE SYSTEMS, 4. *Proceedings...* Sept. 1988. p. 1–24.

[10] ARVIND; IANUCCI, R. A. Two fundamental issues in multiprocessing. In: THAKKAR, S. S. (ed.). *Selected reprints on dataflow and reduction architectures.* IEEE Press, 1987. p.140–164.

[11] ARVIND; NIKHIL, R. S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, v.39, n.3, p.300–318, Mar.1990.

[12] ARVIND; NIKHIL, R. S.; PINGALI, K. K. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, v.11, n.4, p.598–632, Oct.1989.

[13] AUGUST, M. C. Cray X-MP: the birth of a supercomputer. *IEEE Computer*, v.22, n.1, Jan.1989.

[14] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, v.21, n.8, p.613–641, Aug.1978.

[15] BECK, M.; PINGALI, K. K.; NICOLAU, A. Static scheduling for dynamic dataflow machines. *Journal of Parallel and Distributed Computing*, v.10, n.4, p.279–288, Dec.1990.

[16] BHUYAN, L. N.; YANG, Q.; AGRAWAL, D. P. Performance of multiprocessor interconnection networks. *IEEE Computer*, v.22, n.2, p.25–37, Feb.1989.

[17] BRIGGS, F. A.; DAVIDSON, E. S. Organization of semiconductor memories for parallel-pipelined processors. *IEEE Transactions on Computers*, v.22, n.6, p.162–169, June 1977.

[18] BUEHRER, R.; EKANADHAM, K. Incorporating data flow ideas into von Neumann processors for parallel execution. *IEEE Transactions on Computers*, v.36, n.12, p.1515–1522, Dec.1987.

[19] CALSAVARA, C. M. F. R. *Estudo de geração de código para uma máquina de fluxo de dados.* Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, Unicamp, nov.1989.

[20] CATTO, A. J. *Non-deterministic programming in a dataflow environment.* Thesis (Ph.D. in Computer Science), Department of Computer Science, University of Manchester, June 1981.

[21] CHANG, D. Y.; KUCK, D. J.; LAWRIE, D. H. On the effective bandwith of parallel memories. *IEEE Transactions on Computers*, v. 26, n. 5, p. 480–489, May 1977.

[22] CHEN, C.; LIAO, C. Analysis of vector access performance on skewed interleaved memory. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* June 1989. p. 387–394.

[23] CULLER, D. E.; ARVIND. Resource requirements of dataflow programs. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 15. *Proceedings...* June 1988. p. 141–150.

[24] CULLER, D. E. et al. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 4. *Proceedings...* Apr. 1991. p. 164–175.

[25] CULLER, D. E.; PAPADOPOULOS, G. M. The explicit token store. *Journal of Parallel and Distributed Computing*, v. 10, n. 4, p. 289–308, Dec. 1990.

[26] DASGUPTA, S. *Computer architecture: a modern synthesis.* Wiley, 1989. 2 v. V. 2: Advanced Topics.

[27] _____. _____. Wiley, 1989. 2 v. V. 1: Foundations.

[28] DAVIS, A. L.; KELLER, R. M. Data flow program graphs. *IEEE Computer*, v. 15, n. 2, p. 26–41, Feb. 1982.

[29] FENG, T. A survey of interconnection networks. *IEEE Computer*, p. 12–27, Nov. 1981.

[30] FEO, J. T.; CANN, D. C.; OLDEHOEFT, R. R. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, v. 10, n. 4, p. 349–366, Dec. 1990.

[31] FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, v. 21, n. 9, p. 948–960, Sept. 72.

[32] GAJSKI, D. D. et al. A second opinion on data flow machines and languages. *IEEE Computer*, v. 15, n. 2, p. 58–69, Feb. 1982.

[33] GAJSKI, D. D.; PIER, J. Essential issues in multiprocessor systems. *IEEE Computer*, p. 9–27, June 1985.

[34] GAO, G. R.; HUM, H. H. J.; MONTI, J. Towards an efficient hybrid dataflow architecture model. In: PARLE CONFERENCE. *Proceedings...* June 1991. (Lecture Notes in Computer Science, 505). p. 365–371.

[35] GAO, G. R.; HUM, H. H. J.; WONG, Y. Towards efficient fine-grain software pipelining. In: INTERNATIONAL SYMPOSIUM ON SUPERCOMPUTING, 1990. *Proceedings...* June 1990. p. 369–379.

[36] GAUDIOT, J.-L. Structure handling in data-flow systems. *IEEE Transactions on Computers*, v. 35, n. 6, p. 489–501, June 1986.

[37] GHOSAL, D.; BHUYAN, L. N. Performance evaluation of a dataflow architecture. *IEEE Transactions on Computers*, v. 39, n. 5, p. 615–627, May 1990.

[38] GRAFE, V. G. et al. The Epsilon dataflow-processor. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* May 1989. p. 36–45.

[39] GRAFE, V. G.; HOCH, J. E. The Epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Computing*, v. 10, n. 4, p. 309–318, Dec. 1990.

[40] GUPTA, A. et al. Comparative evaluation of latency reducing and tolerating techniques. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 18. *Proceedings...* May 1991. p. 254–263.

[41] GURD, J. R. et al. Fine-grain computing: the dataflow approach. In: *Future parallel computers.* Springer-Verlag, June 1986. (Lecture Notes in Computer Science, 272). p. 82–152.

[42] GURD, J. R.; KIRKHAM, C. C.; WATSON, I. The Manchester prototype dataflow computer. *Communications of the ACM*, v. 28, n. 1, p. 34–52, June 1985.

[43] GURD, J. R.; WATSON, I. Preliminary evaluation of a prototype dataflow computer. In: IFIP WORLD COMPUTING CONGRESS, 9. *Proceedings...* Sept. 1983. p. 545–551.

[44] HALSTEAD, R. H.; FUJITA, T. Masa: a multithreaded processor architecture for parallel symbolic computing. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 15. *Proceedings...* June 1988. p. 443–451.

[45] HIRATA, H. et al. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19. *Proceedings...* June 1992. p. 136–145.

[46] HUM, H. H. J.; GAO, G. R. A novel high-speed memory organization for fine-grain multi-threaded computing. In: PARLE CONFERENCE. *Proceedings...* June 1991. (Lecture Notes in Computer Science, 505). p. 34–51.

[47] HWANG, K.; BRIGGS, F. A. *Computer architecture and parallel processing.* McGraw-Hill, 1985.

[48] IANUCCI, R. A. Toward a dataflow/von Neumann hybrid architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 15. *Proceedings...* June 1988. p. 131–140.

[49] III, D. T. Harper. Reducing memory contention in shared memory multiprocessors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 18. *Proceedings...* May 1991. p. 66–73.

[50] INAGAMI, Y.; FOLEY, J. F. The specification of a new Manchester dataflow machine. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 3. *Proceedings...* June 1989. p. 371–380.

[51] KAWAKAMI, K.; GURD, J. R. A scalable data flow structure store. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 13. *Proceedings...* June 1986. p. 243–250.

[52] KRUATRACHUE, B.; LEWIS, T. Grain size determination for parallel processing. *IEEE Software*, p. 23–32, Jan. 1988.

[53] KURIAN, L.; HERLINA, P. T.; CORAOR, L. D. Memory latency effects in decoupled architectures with a single data memory module. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19. *Proceedings...* June 1992. p. 236–245.

[54] LEE, D. Scrambled storage for parallel memory systems. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 15. *Proceedings...* June 1988. p. 232–239.

[55] LORENZO, P. A. R.; CATTO, A. J. Escalonamento de processos considerando atrasos de comunicação. In: SIMPóSIO BRASILEIRO DE AR-

QUITETURA DE COMPUTADORES — PROCESSAMENTO DE ALTO DESEMPENHO, 4. *Anais...* out. 1992. p. 537–545.

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

[57] NIKHIL, R. S.; ARVIND. Can dataflow subsume von Neumann computing? In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* May 1989. p. 262–273.

[58] NIKHIL, R. S.; PAPADOPOULOS, G. M.; ARVIND. *T: a multithreaded massively parallel architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19. *Proceedings...* June 1992. p. 156–167.

[59] PAPADOPOULOS, G. M. *Implementation of a general-purpose dataflow multiprocessor.* Cambridge: MIT Press, 1991.

[60] PAPADOPOULOS, G. M.; CULLER, D. E. Monsoon: an explicit token-store architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 17. *Proceedings...* May 1990. p. 82–91.

[61] PAPADOPOULOS, G. M.; TRAUB, K. R. Multithreading: a revisionist view of dataflow architectures. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 18. *Proceedings...* May 1991. p. 342–351.

[62] PATNAIK, L. M.; GOVINDARAJAN, R.; RAMADOSS, R. Design and performance evaluation of EXMAN: An EXtended MANchester data flow computer. *IEEE Transactions on Computers*, v. 35, n. 3, p. 229–243, Mar. 1986.

[63] PATTERSON, D. A. Reduced instruction set computers. *Communications of the ACM*, v. 28, n. 1, p. 8–21, Jan. 1985.

[64] RAU, B. R. et al. The Cydra 5 departmental supercomputer. *IEEE Computer*, v. 22, n. 1, p. 12–35, Jan. 1989.

[65] REED, D. A.; GRUNWALD, D. C. The performance of multicomputer interconnection networks. *IEEE Computer*, v. 20, n. 6, p. 63–73, June 1987.

[66] RETTBERG, R.; THOMAS, R. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, v. 25, n. 12, p. 1202–1212, Dec. 1986.

[67] RUGGIERO, C. A.; SARGEANT, J. Control of paralelism in the Manchester dataflow computer. In: CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 3. *Proceedings...* 1987. (Lecture Notes in Computer Science, 274). p. 1–15.

[68] RUSSEL, C. H.; WATERMAN, P. J. Variations on UNIX for parallel-processing computers. *Communications of the ACM*, v. 30, n. 12, p. 1048–1055, Dec. 1987.

[69] 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.

[70] SAKAI, S. et al. An architecture of a dataflow single chip processor. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* May 1989. p. 46–53.

[71] SARGEANT, J.; KIRKHAM, C. C. Stored data flow structure store. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 13. *Proceedings...* June 1986. p. 235–242.

[72] SATO, M. et al. Thread-based programming for the EM-4 hybrid dataflow-machine. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19. *Proceedings...* June 1992. p. 146–155.

[73] SCHAUSER, K. E.; CULLER, D. E.; EICKEN, T. von. Compiler-controlled multithreading for lenient parallel languages. In: CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 5. *Proceedings...* Aug. 1991. (Lecture Notes in Computer Science, 523). p. 50–72.

[74] SEZNEE, A.; LENFANT, J. Interleaved parallel schemes: improving memory throughput on supercomputers. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19. *Proceedings...* June 1992. p. 246–255.

[75] SHAPIRO, H. D. Theoretical limitations on the efficient use of parallel memories. *IEEE Transactions on Computers*, v. 27, n. 5, p. 421–428, May 1978.

[76] SILVA, S. R. P. da. *Modelamento e análise de uma arquitetura dirigida pelo fluxo de dados.* Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, Unicamp, 1991.

[77] SMITH, B. J. Architecture and applications of the HEP multiprocessor computer system. In: SPIE. *Proceedings...* 1981. p.241–248.

[78] STEMSTRÖM, P. Reducing contention in shared-memory multiprocessors. *IEEE Computer*, v.21, n.11, p.26–37, Nov.1988.

[79] _____. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, v.23, n.6, p.12–24, June 1990.

[80] STONE, H. S. *High-performance computer architecture.* Addison-Wesley, 1987.

[81] TRAUB, K. Multi-threaded code generation for dataflow architectures from non-strict programs. In: CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 5. *Proceedings...* Aug. 1991. (Lecture Notes in Computer Science, 523). p.73-101.

[82] TRELEAVEN, P. C.; BROWNBRIDGE, D. R.; HOPKINS, R. P. Data-driven and demand-driven computer architecture. *ACM Computing Surveys*, v.14, n.1, p.93–143, Mar.1982.

[83] VEEN, A. H. Dataflow machine architecture. *ACM Computing Surveys*, v.18, n.4, p.365–396, Dec.1986.

[84] VISOLI, M. C. *Exploração de seqüencialidade em fluxo de dados.* Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, Unicamp, (em preparação).

[85] VISOLI, M. C.; CATTO, A. J. Tratamento de código seqüencial no modelo de fluxo de dados. In: SIMPóSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES — PROCESSAMENTO DE ALTO DESEMPENHO, 4. *Anais...* out.1992. p.147–158.

[86] WATSON, I.; GURD, J. R. A practical dataflow computer. *IEEE Computer*, v.15, n.2, p.51–57, Feb.1982.

[87] WEBER, W.; GUPTA, A. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* June 1989. p.273–280.

[88] WEISS, S. An aperiodic storage scheme to reduce memory conflicts in vector processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 16. *Proceedings...* June 1989. p.380–386.

[89] YAMAGUCHI, Y. Synchronization mechanisms of a highly parallel dataflow machine EM-4. *IEICE Transactions*, v. E74, n. 1, Jan. 1991.

[90] YAMANA, H. et al. Parallel processing system Harray. *Computer Systems in Engineering*, v. 1, n. 1, p. 111-130, Jan. 1990.

[91] 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.

[92] YUBA, T. et al. Dataflow computer development in Japan. In: INTERNATIONAL SYMPOSIUM ON SUPERCOMPUTING, 1990. *Proceedings...* June 1990. p. 140–147.