014683-2

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# IMPLEMENTAÇÃO DE ARQUITETURAS SIMD

por

Alexandre da Silva Carissimi

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

Prof. Philippe O. A. Navaux
Orientador

Porto Alegre, Setembro de 1989.

UFRGS
BIBLIOTECA
CPD/PGCC

CATALOGAÇÃO NA FONTE

Carissimi, Alexandre da Silva
Implementação de Arquiteturas SIMD
Porto Alegre, PGCC da UFRGS, 1989.
196 p.
Diss. (mestr. ci. comp.) UFRGS-PGCC, Porto Alegre, BR-RS, 1989.

Dissertação: Arquitetura de Computadores: Processamento Paralelo: Algoritmos Paralelos.

U F R G S
BIBLIOTECA
CPD/PGCC

"O homem está disposto a morrer por uma idéia sempre que dela não tem uma idéia muito clara"

Gilbert Keith Cherstenton (1874-1936)
Escritor Inglês

U F R G S
BIBLIOTECA
CPD/PGCC

Aos meus pais,

Lélio (In Memoriam),

e Nelcy.

UFRGS
BIBLIOTECA
CPD/PGC

# AGRADECIMENTOS

Agradeço aos orgãos governamentais, CAPES e CNPq, pelo auxílio financeiro recebido em forma de bolsa durante a minha permanência no curso de pós-graduação em computação. Em particular a todos aqueles que apesar de todos os problemas de falta de verbas, políticos e sociais, lutam e acreditam neste País, na sua soberania, e no seu potencial humano e científico.

Ao meu amigo e "mestre" Philippe O. A. Navaux, que por uma coincidência é Professor Doutor nesta universidade e meu orientador, pela amizade, companheirismo, dedicação, incentivo e ensinamentos transmitidos a este "discípulo".

A todo o corpo docente do CPGCC pela importante contribuição que deram para o meu crescimento como pessoa e como profissional.

As meninas da secretaria, e aos demais funcionários do CPGCC e DI, pela forma com a qual fui tratado durante estes anos.

A turma da biblioteca do CPGCC pela gentileza, presteza, atenção e dedicação com que sempre fui atendido.

Ao Eng. Msc. Celso Luis Mendes, do Instituto Nacional de Pesquisas Espaciais - INPE, pela amizade e pelo apoio a mim dado em forma de material sobre o GAPP, discussões e idéias.

Ao colega e amigo, Eduardo "zague" Todt pelo companheirismo, e pelas reuniões, discussões e sugestões recebidas.

Ao colega e amigo João Cesar Netto pela amizade e companheirismo.

A todos os colegas do CPGCC, aos companheiros dos jogos de futebol, de tênis, de chimarrão, de festas & jantares, almoços, e viagens, pelo companheirismo.

A todas as demais pessoas, que das mais diferentes formas contribuiram direta e indiretamente na minha vida e no meu trabalho, em especial aos meus irmãos André, Leandro e Leonardo.

A minha mãe Nelcy pelo incansável carinho, amor, dedicação com que me educou, por todo o apoio que recebi e recebo, pelos ensinamentos e pelo exemplo de garra e força no enfrentar os. obstáculos que a vida nos apresenta. Obrigado, mãe!

Ao meu pai Lélio por todo o incansável esforço que dedicou durante toda vida a minha educação e a dos meus irmãos, possibilitando com que hoje eu escrevesse este agradecimento em uma dissertação de mestrado. Pelo exemplo de justiça, de dedicação a tudo, de "dureza", de pai, de profissional, de amigo, em que transformaste tua vida. Por todo carinho, amor, compreensão e ensinamentos recebidos. Obrigado, pai!

A todas as dificuldades enfrentadas pelo prazer de superá-las.

E a você...

SUMÁRIO

GLOSSÁRIO .......... 13
LISTA DE ABREVIATURAS .......... 15
LISTA DE FIGURAS .......... 17
RESUMO .......... 19
ABSTRACT .......... 21

1 INTRODUÇÃO .......... 23

1.1 Classificação de arquiteturas para processamento paralelo .......... 25
1.2 Arquitetura SIMD - processadores matriciais... 26
1.2.1 Organização de processadores matriciais. 27
1.2.2 Organização de Processadores matriciais associativos .......... 29
1.2.2.1 Organização totalmente paralela. 29
1.2.2.2 Organização serial por bit .......... 32
1.3 Objetivos do presente trabalho .......... 33

2 MAPEAMENTO DE ALGORITMOS EM ARQUITETURAS PARALELAS 35
2.1 Obtenção de algoritmos paralelos .......... 38
2.2 Mapeamento de algoritmos .......... 39
2.3 Avaliação de algoritmos paralelos .......... 43
2.4 Linguagens voltadas a exploração do paralelismo .......... 44
2.5 Restrições a algoritmos para arquiteturas SIMD .......... 45

3 PROCESSAMENTO DE IMAGENS: UM EXEMPLO DE APLICAÇÃO. 49

3.1 Caracterização do problema processamento de imagens .......... 50
3.2 Arquiteturas paralelas para processamento de imagens .......... 52
3.3 Algoritmos para processamento de imagens .......... 54
3.3.1 Mapeamento de níveis de cinza .......... 54
3.3.2 Filtragem .......... 56

3.4 Implementação do algoritmo de Sobel em arquiteturas SIMD ........................ 61
3.5 Considerações sobre aplicações em máquinas SIMD ........................................ 66

4 PROPOSTA PARA ARQUITETURA PARALELA MATRICIAL...... 71

4.1 Arquitetura genérica para processadores matriciais .......................... 72
4.2 Arquitetura do sistema proposto .............. 74
4.2.1 Máquina hospedeira ...................... 76
4.2.2 Matriz de processadores ................ 76
4.2.3 Interface com o hospedeiro ............ 78
4.2.4 Módulo de controle ...................... 80
4.2.5 Módulo de memória de programa .......... 82
4.2.6 Módulo de reformatação de dados ........ 84
4.3 Interpretador - O programa de controle ....... 91
4.3.1 Estruturas de memória ................. 91
4.3.2 Registradores internos ................. 92
4.3.3 Implementação do interpretador ......... 95

5 AMBIENTE PARA DESENVOLVIMENTO DE PROGRAMAS ....... 99

5.1 Compilador GAL - CG .......................... 100
5.2 CORNER.SIF - O programa de E/S .............. 102
5.3 Carregador ................................... 104
5.4 Desenvolvimento de programas no sistema proposto ..................................... 106

6 VALIDAÇÃO E CONSIDERAÇÕES DE DESEMPENHO .......... 111

6.1 Validação funcional da arquitetura proposta .. 111
6.2 Considerações de desempenho .................. 115
6.2.1 Tempo de E/S ........................... 116
6.2.2 Complexidade das operações ............ 120
6.2.3 Organização de dados em memória e comunicação entre processadores ............. 123

7 CONCLUSÃO ...................................... 129

ANEXO 1 - Descrição do GAPP (Geometric Arithmetic Parallel Processor)........................ 133

ANEXO 2 - Linguagem GAL (GAPP Algorithmic Language).. 139

ANEXO 3 - Compiler GAL commands ...................... 143

ANEXO 4 - The generator/interpreter model opcodes ... 147

ANEXO 5 - The intermediate output file format ....... 155

ANEXO 6 - Listagem do simulador ..................... 163

ANEXO 7 - Programas exemplos e simulação ............ 177

REFERÊNCIAS BIBLIOGRÁFICAS........................... 191

UFRGS
BIBLIOTECA
CPD/PGCC

# GLOSSÁRIO

| | |
|---|---|
| AND | - operador lógico "E". |
| ASSEMBLY | - linguagem de montagem. |
| BORROW | - transporte negativo (empréstimo) ocorrido durante uma operação de subtração. Inverso de carry. |
| CARRY | - dígito produzido na soma de dois ou mais algarismos, quando o total for igual ou menor que a base do sistema de notação em que os algarismos são representados. |
| DATA FLOW | - tipo de máquina cujo controle de execução é orientado ao fluxo de dados. |
| GENERAL PURPOSE | - computador de uso (aplicação) geral. |
| NOT | - operador lógico de negação. |
| OR | - operador lógico "OU". |
| OVERHEAD | - resultado, ou desempenho, abaixo do esperado de um dispositivo de processamento de dados, ou de um programa. |
| PRE-FETCH | - busca, recuperação antecipada, para obter determinada quantidade de dados de um local de armazenamento. |
| SORTING | - procedimento de ordenação de dados a partir da aplicação de determinada regra sobre estes. |
| TIME SHARING | - pertencente, ou relativo, ao uso concorrente do tempo em um dispositivo. |
| TRI STATE | - estado de alta impedância. |
| XOR | - operação lógica "OU exclusivo". |

VLSI - Very Large Scale Integration.

# LISTA DE ABREVIATURAS

| | |
|---|---|
| CAD/CAM | - Computer Aided Design/Computer Aided Manufacturing. |
| CAPES | - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior. |
| CG | - Compiler GAL. |
| CNPq | - Conselho Nacional de Desenvolvimento Científico e Tecnológico. |
| CPGCC | - Curso de Pós Graduação em Ciência da Computação. |
| CPU | - Central Processing Unit. |
| CTLB | - Corner Turning Line Buffer. |
| DI | - Departamento de Informática. |
| DOS | - Disk Operational System. |
| DPCM | - Differential Pulse Code Modulation. |
| E/S | - Entrada e Saída. |
| FFT | - Fast Fourier Transformer. |
| GAL | - GAPP Algorithm language. |
| GAPP | - Geometric Arithmetic Parallel Processor. |
| GO | - Global Output. |
| IBM | - International Business Machine. |
| INPE | - Instituto Nacional de Pesquisas Especiais. |
| LSI | - Large Scale Integration. |
| MIMD | - Multiple Instruction Multiple Data. |
| MISD | - Multiple Instruction Single Data. |
| MOPS | - Milhões de Operações por Segundo. |
| PC | - Personal Computer. |
| PE | - Processing Element. |
| RAM | - Random Acess Memory. |
| RISC | - Reduction Instruction Set Computer. |
| ROM | - Read Only Memory. |
| SIMD | - Single Instruction Multiple Data. |
| SISD | - Single Instruction Single Data. |
| TO | - Timer 0 do microprocessador 8031. |
| T1 | - Timer 1 do microprocessador 8031. |
| ULA | - Unidade Lógica Aritmética. |

LISTA DE FIGURAS


Figura 1.1 - Organizações para Processadores Matriciais ... 28
Figura 1.2 - Organização totalmente paralela ... 30
Figura 1.3 - Organização totalmente paralela com lógica distribuída ... 31
Figura 2.1 - Etapas no desenvolvimento de arquiteturas paralelas ... 36
Figura 2.2 - Esquema para Classificação de arquiteturas paralelas ... 41
Figura 2.3 - Características de algoritmos e arquiteturas ... 42
Figura 3.1 - Cone de Processamento ... 52
Figura 3.2 - Exemplos de funções para mapeamento de niveis de cinza ... 55
Figura 3.3 - Máscara para operação de suavização ... 57
Figura 3.4 - Máscara para detecção de bordas ... 58
Figura 3.5 - Máscara para operação de Laplaciano ... 58
Figura 3.6 - Máscara para operação de Sobel ... 60
Figura 3.7 - Opções de Interconexão ... 62
Figura 3.8 - Organização de Memória ... 63
Figura 3.9 - Avaliação de Ys ... 64
Figura 4.1 - Arquitetura genérica para máquinas matriciais ... 73
Figura 4.2 - Arquitetura do sistema proposto ... 74
Figura 4.3 - Conexão hospedeiro-processador matricial.. 76
Figura 4.4 - Organização da matriz de processadores ... 77
Figura 4.5 - Registradores de interface ... 79
Figura 4.6 - Estrutura do módulo de controle ... 82
Figura 4.7 - Estrutura do módulo de memória ... 83
Figura 4.8 - "corner turning" com conexão cilíndrica .. 85
Figura 4.9 - Algoritmo de "corner turning" por deslocamento ... 86
Figura 4.10 - Programa em GAL para "corner turning" por deslocamento ... 86


UFRGS BIBLIOTECA

Figura 4.11 - Estrutura do módulo de reformatação de dados ... 89
Figura 4.12 - Adaptação da rotina de "Corner Turning" para o sistema proposto ... 90
Figura 5.1 - Organização do Compilador GAL ... 101
Figura 5.2 - Mapeamento de memória ... 105
Figura 5.3 - Estrutura geral de um programa para o sistema proposto... 107
Figura 5.4 - Exemplo de utilização do processador matricial ... 108
Figura 6.1 - Implementação do algoritmo de mapeamento em níveis de cinza ... 117
Figura 6.2 - Tempos de execução T(E/S)+T(e) (em ciclos) 119
Figura 6.2a - Imagem 16 x 16 ... 119
Figura 6.2b - Imagem 32 x 32 ... 119
Figura 6.2c - Imagem 64 x 64 ... 119
Figura 6.3 - Tempo de execução das operações aritméticas elementares (em ciclos)... 121
Figura 6.3a - Sinal magnitude ... 121
Figura 6.3b - Complemento de dois ... 121
Figura 6.3c - Sem sinal ... 121
Figura 6.4 - Implementação GAL do algoritmo de Sobel .. 124
Figura A.1 - Interconexão entre processadores ... 134
Figura A.2 - Arquitetura interna do elemento processador ... 135
Figura A.3 - Operações lógicas ... 136
Figura A.4 - Conjunto de instruções ... 137

# RESUMO

Este trabalho descreve a área de processamento matricial, mostrando os principais compromissos existentes na obtenção de arquiteturas paralelas a partir de algoritmos, para que haja um ganho real na avaliação destes. São feitas, ainda, considerações sobre ferramentas de programação para arquiteturas paralelas.

Os principais compromissos que influenciam as arquiteturas SIMD, objeto de estudo deste trabalho, são abordados analisando-se uma área de aplicação de arquiteturas SIMD: tratamento de imagens.

Como um caso prático de estudo e exemplo destes compromissos, é proposta uma arquitetura SIMD para um processador matricial empregando um chip matricial disponível comercialmente - o GAPP (Geometric Arithmetic Parallel Processor). É proposto, ainda, um ambiente para o desenvolvimento de programas nesta arquitetura. Este ambiente é baseado na utilização da linguagem GAL (GAPP Algorithm Language), criada especificamente para elaboração de programas para o GAPP.

## ABSTRACT

This work describes the array processing area, discussing the main tradeoffs in the design of parallel architectures from algorithms. The algorithm to architecture transformation is called a mapping problem. Some considerations about programming tools for parallel architectures are also made.

The relationship between algorithms and architectures is covered by studying a specific case for SIMD architectures: digital image processing.

A SIMD architecture proposal, using a commercially available chip array - GAPP (Geometric Arithmetic Parallel Processor) is made. This architecture is used on a practical case to study and analyze those tradeoffs. An environment for program development for this architecture is also proposed. This environment is based on the use of GAL language (GAPP Algorithm Language), which was created specificaly for GAPP program development.

UFRGS
BIBLIOTECA

# 1 INTRODUÇÃO

A utilização de computadores digitais para a solução de problemas científicos é cada vez mais uma constante em todas as áreas do conhecimento. Razões de custo e viabilidade, na execução de experimentos em laboratórios, levam a necessidade de modelos matemáticos para a avaliação e análise de novas teorias e no aperfeiçoamento das já existentes. O emprego de modelos apresenta, ainda, como vantagens a possibilidade de obter-se resultados mais precisos que em experimentos físicos, e uma maior flexibilidade na realização de "ajustes finos" da teoria, não apresentando restrições físicas de implementação - como necessidade de temperaturas controladas, vácuo, atrito - a não ser a própria capacidade de velocidade e memória da máquina empregada na simulação.

Além do emprego em pesquisas puramente científicas, a utilização de computadores digitais também está vinculada a atividades econômicas, representando uma parcela significativa no desenvolvimento sócio-econômico de uma nação. Aplicações na área de CAD/CAM, em indústrias dos mais diversos setores, e a exploração de recursos minerais a partir de imagens de satélites são alguns exemplos desta importância.

Uma característica frequentemente encontrada na utilização de computadores nestas aplicações é a manipulação de grandes quantidades de dados, implicando na necessidade de altas taxas de processamento. Existem aplicações cujos tempos de computação, mesmo para as máquinas mais modernas, inviabilizam uma análise em tempos considerados razoáveis. A área de processamento paralelo, a partir do uso de supercomputadores e de arquiteturas não convencionais (paralelas), surge como uma alternativa a suprir esta necessidade.

A busca do paralelismo em avaliações computacionais é realizada em duas linhas distintas de máquinas: as uniprocessadas e as pluriprocessadas. As máquinas uniprocessadas são aquelas que seguem a organização proposta por Von Neumann. Basicamente, nestes sistemas, tenta-se obter paralelismo através do emprego de organizações hierárquicas de memória (cache), com a sobreposição de operações que empregam a CPU com operações de E/S, com a utilização de "pre-fetch", ou ainda através de multiprogramação e "time sharing".

As máquinas paralela, também denominadas de computadores paralelos, são sistemas que executam explicitamente avaliações computacionais em paralelo [HWA 84]. O paralelismo, aqui, pode ser atingido, entre outras formas, através da replicação completa de unidades de processamento, na divisão e alocação de recursos, na exploração de características paralelas inerentes a determinado algoritmo ou problema.

Em função do tipo de paralelismo empregado, os computadores paralelos são divididos em três grupos: máquinas pipeline, processadores matriciais e multiprocessadores. As máquinas pipeline superpoêm a execução de instruções explorando um paralelismo denominado de **temporal**. Processadores matriciais empregam a multiplicidade de unidades processadoras executando instruções sincronamente utilizando o paralelismo dito **espacial síncrono**. Por último, temos o paralelismo **espacial assíncrono**, que é empregado em sistemas multiprocessadores através da divisão de recursos comuns (memórias, periféricos etc..) por um conjunto de processadores. Convém salientar que estas três abordagens não são mutuamente exclusivas.

## 1.1 Classificação de arquiteturas para processamento paralelo

A necessidade crescente da utilização de computadores de alto desempenho, nas mais diversas áreas de aplicação, incentivou pesquisas na área de processamento paralelo. Muitas máquinas com arquiteturas alternativas foram propostas nas últimas décadas como resultados destas pesquisas. A diversidade de formas empregadas na exploração de paralelismo gerou a necessidade da criação de uma metodologia para organizar e classificar estes novos tipos de arquiteturas. Como exemplo, citam-se as classificações propostas por M. Flynn [FLY 72], T. Feng [FEN 72], e W. Hänler [HAN 77]. Neste trabalho irá adotar-se a classificação proposta por Flynn por ser a mais comumente empregada.

A classificação proposta por Flynn leva em conta a forma pela qual é executada uma instrução em um conjunto de dados. Segundo Flynn as máquinas são organizadas em quatro categorias, obtidas a partir do produto cartesiano de (Single Instruction, Multiple Instruction) X (Single Data, Multiple Data), abreviadas por (SI,MI) X (SD,MD). As quatro categorias são, então:

- Single Instruction stream Single Data stream (SISD);
- Single Instruction stream Multiple Data stream (SIMD);
- Multiple Instruction stream Single Data stream (MISD);
- Multiple Instruction stream Multiple Data stream (MIMD).

Os processadores matriciais, objeto de estudo do presente trabalho, são classificados neste esquema como máquinas SIMD.

## 1.2 Arquiteturas SIMD - processadores matriciais

Um processador matricial é um computador paralelo síncrono com múltiplas unidades lógicas e aritméticas (ULA), denominados de elementos processadores (PE - do inglês Processing Elements) que podem operar em paralelo de forma síncrona realizando uma mesma função em um mesmo instante [HWA 84]. O tipo de paralelismo explorado pelos processadores matriciais é o espacial, devido a sua característica de multiplicidade de ULAs.

Os elementos processadores que compoẽm o processador matricial podem ser interligados entre si utilizando diversas redes de interconexão, pode-se citar como exemplo, a interconexão mesh, barrel shift, cubo e shuffle. Não é escopo deste trabalho abordar e discutir as diversas redes de interconexão, pode-se, porém, enumerar uma série de trabalhos no assunto como [BHU 87], [FEN 81] e [THU 74].

Existe, ainda, uma derivação dos processadores matriciais que são os processadores denominados associativos. Estes diferem dos primeiros na forma pela qual empregam para acesso e operação sobre os dados. Enquanto os processadores matriciais utilizam o endereço do dado para acesso, os matriciais associativos exploram a capacidade de memórias associativas acessando os dados via conteúdo.

Um processador matricial, normalmente, está vinculado a um computador hospedeiro através de uma lógica de controle responsável por este interfaceamento. O computador hospedeiro, geralmente, é um sistema "general purpose" sendo classificado, segundo Flynn, como uma máquina SISD. A função principal do computador hospedeiro é o gerenciamento de operações de E/S, interagindo o processador matricial com o meio externo.

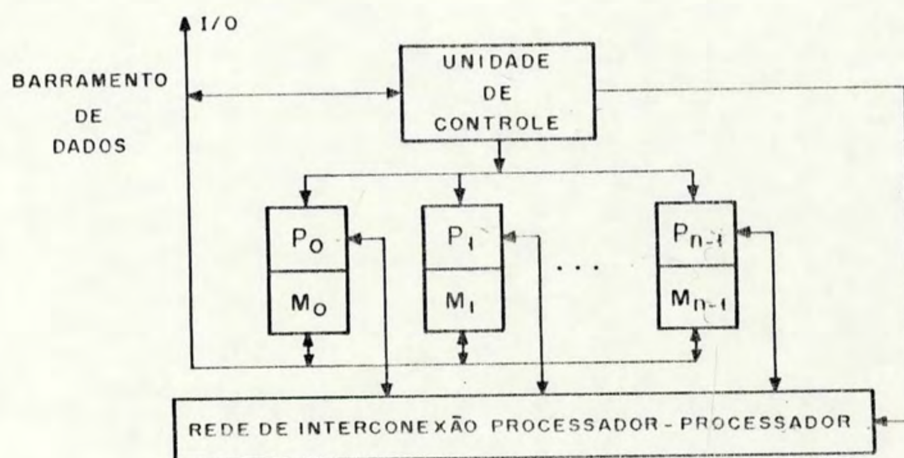
### 1.2.1 Organização de processadores matriciais

Os processadores matriciais são compostos por uma unidade de controle, um conjunto de elementos processadores, uma memória associada a cada elemento processador e uma rede de interconexão. Estes elementos básicos são configurados em duas organizações levando em conta a maneira pela qual a memória é organizada. Em ambas organizações a unidade de controle é responsável pelo gerenciamento da operação dos elementos processadores decodificando as instruções a serem executadas e gerando os sinais de controle.

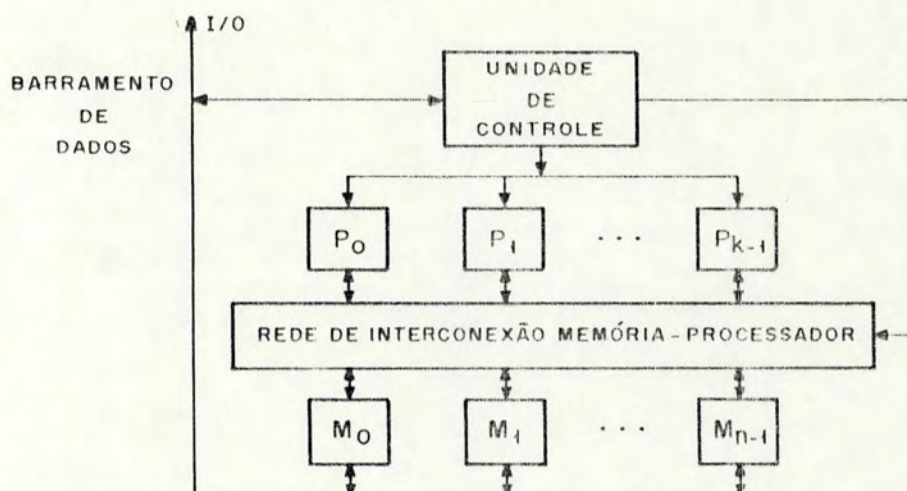
Na primeira organização (figura 1.1a) os elementos processadores são compostos por uma memória local e um conjunto de registradores de trabalho, utilizados na execução das operações. Os elementos processadores são interligados entre si através de uma rede de interconexão que pode ser configurada via unidade de controle (rede dinâmica), ou construtivamente (rede estática). Os dados a serem operados são enviados às memórias locais de cada elemento processador antes da execução das instruções em paralelo.

A segunda forma (figura 1.1b) de organizar-se a arquitetura de um processador matricial difere da primeira em dois aspectos básicos. As memórias locais associadas a cada elemento de processamento são substituídas por módulos de memória comuns a todos os elementos processadores. Segundo, a rede de interconexão entre os elementos processadores é substituída por uma estrutura de conexão módulo de memória - elemento processador. O controle desta interconexão, vinculando memória a elemento de processamento é determinada pela unidade de controle.

UFRGS
BIBLIOTECA
CPD/PGCC




(a)




(b)

figura 1.1 - Organizações para processadores matriciais

### 1.2.2 Organização de processadores matriciais associativos

Os processadores matriciais associativos são compostos por vários elementos processadores organizados de forma análoga aos processadores matriciais, porém empregam memórias endereçáveis por conteúdo as quais possibilitam a recuperação de informações de forma associativa. Os processadores matriciais associativos são classificados, segundo o processo de comparação empregado, em duas categorias:

- totalmente paralela ("fully parallel");
- serial por bit ("bit serial").

#### 1.2.2.1 Organização totalmente paralela

As arquiteturas associativas totalmente paralelas empregam como método de comparação uma lógica associada a cada bit das palavras que compõem a memória. Esta categoria é subdividida, levando-se em conta a maneira pela qual a decisão lógica é fornecida, em: organizado a palavra ("word organized"), e lógica distribuída ("distributed logic").

Uma arquitetura totalmente paralela organizada a palavra possui a informação de decisão associada a cada palavra. Esta organização implica em poder-se pesquisar um conteúdo em toda a memória, em um único ciclo, obtendo-se palavras que cumpram uma determinada condição. Devido a esta característica diz-se que este tipo de arquitetura opera de uma forma paralela por palavra ("parallel by word").

A maior vantagem deste tipo de organização é a sua simplicidade e rapidez, já que as operações são executadas em paralelo em toda a memória ao mesmo tempo. A desvantagem principal é a quantidade de hardware necessário para implementá-las, pois a cada bit de todas palavras há necessidade de uma lógica para executar comparação.

A arquitetura de um processador associativo to-

talmente paralelo pode ser esquematizado como na figura 1.2. Cada ponto no cruzamento de linhas e colunas representa a existência de uma lógica de comparação. A lógica de mascaramento atua selecionando porções da palavra a ser comparada, habilitando e desabilitando a lógica de comparação associada a cada bit da palavra. A lógica de decisão é responsável por operar os resultados parciais de cada lógica de comparação compondo um resultado booleano, verdadeiro ou falso, fornecendo ao meio externo quais as palavras que cumprem ou não uma determinada condição de pesquisa.

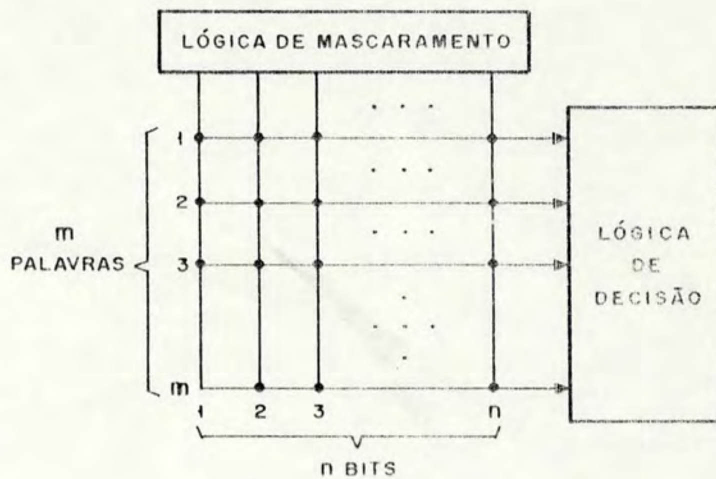



figura 1.2 - Organização totalmente paralelas

Para as arquiteturas totalmente paralelas com lógica distribuída o resultado do procedimento de pesquisa não é associado a palavra, mas sim a um grupo de bits, denominado de célula de caracter. Esta organização é composta de uma unidade de controle e um conjunto de células de caracter, cada uma das quais armazena um caracter de informação. Cada célula é formada por duas partes: a informação propriamente dita e um controle de estado atual, podendo comunicar-se com as duas células vizinhas a si e com a unidade de controle. Cada célula avalia a condição de entrada (fornecida por um barramento de dados e controle) independentemente e em paralelo. Em função do resultado obtido pode-se, tipicamente, ignorar ou realizar determinada operação, trocar de estado, transmitir informações a célula vizinha, ler ou escrever dados no barramento. Este conceito foi desenvolvido por Lee [LEE 63] e apresenta facilidade para recuperação de informações. A figura 1.3 fornece um esquema para esta organização.

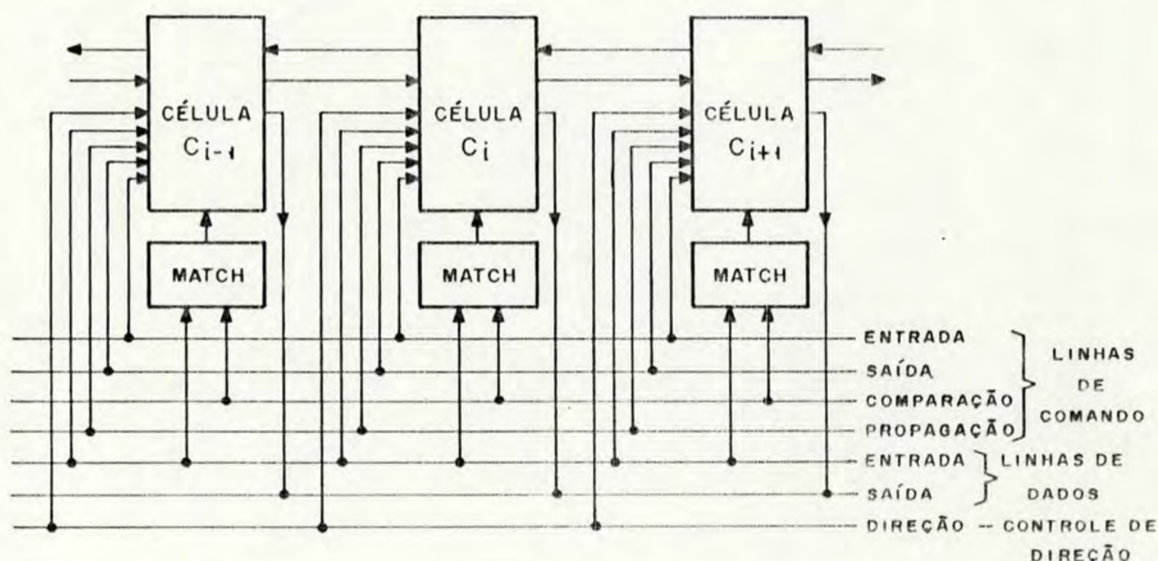



figura 1.3 - Organização Totalmente Paralela com lógica distribuída

### 1.2.2.2 Organização serial por bit

Os processadores associativos seriais por bit foram inicialmente propostos por Shomman [SHO 60], como uma alternativa para a redução do hardware necessário a implementação de arquiteturas associativas totalmente paralelas. A arquitetura básica dos processadores associativos seriais por bit é composta por uma lógica de seleção de bit, uma lógica de mascaramento, uma lógica de comparação e decisão, e uma unidade de controle. A diferença fundamental desta organização, em relação à totalmente paralela, é que a lógica de comparação é associada a cada palavra que compõe a memória, e não mais a cada bit. O procedimento de operação desta organização é o que segue.

Analogamente aos processadores associativos totalmente paralelos a lógica de mascaramento é responsável pela seleção de porções da palavra para ser efetuada a operação de comparação. Um determinado bit, de todas palavras, é habilitado pela lógica de seleção de bit. O bit selecionado é então operado pela lógica de comparação e decisão. Este procedimento é repetido tantas vezes quanto necessário, em função do tamanho da palavra a ser comparada. A unidade de controle é responsável por este gerenciamento. Os resultados parciais, obtidos para cada bit, são armazenados na lógica de comparação e decisão, que ao final da operação fornece um resultado booleano indicando quais as palavras que satisfazem a condição de pesquisa (comparação). Devido ao fato das palavras que compoêm a memória serem pesquisadas em paralelo, um bit a cada instante, este tipo de procedimento é denominado de serial por bit paralelo por palavra ("bit serial word parallel").

UFRGS
BIBLIOTECA
CPD/PGCC

## 1.3 Objetivos do presente trabalho

O objetivo do presente trabalho é apresentar a idéia do processamento matricial analisando desde os compromissos entre este modelo computacional e sua aplicação, para que haja um real ganho de desempenho na avaliação, até considerações sobre desenvolvimento de uma arquitetura matricial e aspectos de programação. É apresentado, como objeto de estudo, a proposta de uma arquitetura SIMD matricial empregando-se um chip matricial disponível comercialmente - GAPP (Geometric Arithmetic Parallel Processor) - fabricado pela NCR Corporation [NCR 85d]. Na proposta da arquitetura considerar-se-á a utilização da linguagem de programação GAL (GAPP Algorithm Language) [NCR 85a], desenvolvida especificamente para fornecer suporte de programação para o chip GAPP.

O capítulo 2 aborda os principais compromissos existentes para a obtenção de arquiteturas a partir de algoritmos com características paralelas. Inicialmente, situa-se este problema de uma forma genérica, apresentando os trabalhos que estão sendo desenvolvidos nesta área. A seguir os principais fatores que influenciam as arquiteturas do tipo matricial são considerados. Ainda neste capítulo são feitas considerações sobre linguagens de programação para arquiteturas não convencionais.

O capítulo 3 apresenta um exemplo de área de aplicação do processamento matricial - tratamento digital de imagens - abordando suas características favoráveis ao emprego das arquiteturas SIMD. Apresenta-se, ainda, a influência dos compromissos levantados no capítulo 2 na escolha da aplicação e na definição de uma arquitetura paralela.

No capítulo 4 é proposta uma arquitetura para um processador matricial, empregando o chip GAPP.

As ferramentas de programação necessárias ao

desenvolvimento de programas no sistema proposto são apresentadas no capítulo 5. São abordados, neste, o compilador GAL e os programas de E/S e carregador.

O capitulo 6 apresenta a validação funcional da arquitetura do processador matricial proposto no capítulo 4. Esta validação é executada verificando-se os comandos enviados para a matriz de processadores, via saída de um simulador da parte de controle da arquitetura proposta. São realizados, ainda, considerações a respeito dos compromissos enumerados no capítulo 2, exemplificando-os na arquitetura proposta.

# 2 MAPEAMENTO DE ALGORITMOS EM ARQUITETURAS PARALELAS

Uma arquitetura paralela, geralmente, é um sistema de propósito específico, otimizado para realizar de forma eficiente uma certa gama de funções dentro de uma aplicação. As arquiteturas não convencionais, de uma forma geral, são caracterizadas por não serem versáteis, e por possuirem um alto custo de implementação, justamente devido a esta sua especialização.

Com o avanço tecnológico ocorrido na área de concepção, em especial com o VLSI, o custo de implementação de sistemas digitais foi reduzido, incentivando a criação de diferentes tipos de arquiteturas. A maior vantagem, porém, no emprego de VLSI não está somente na obtenção de uma maior densidade de integração, mas sim na criação de novas arquiteturas que possam explorar características de implementação em VLSI.

O alto desempenho a ser obtido em arquiteturas VLSI é resultado da operação concorrente de um grande número de processadores simples, ao invés do emprego de um único processador por demais especializado. Esta caracterísitica é desejável em função de fatores de simplicidade e regularidade que facilitam, através da replicação de estruturas, a implementação em VLSI. Pode-se citar, como exemplo de estruturas paralelas com condições favoráveis à implementação em VLSI, as arquiteturas sistólicas propostas por Kung [KUN 82].

Surge, porém, um problema, não trivial, na obtenção de arquiteturas paralelas [OFF 85]: como traduzir de forma eficiente um algoritmo computacional em uma implementação física. Esta tradução envolve, no mínimo, a passagem por quatro estágios: uma especificação formal da aplicação, uma representação implícita ou explícita do algoritmo correspondente à aplicação, uma arquitetura computacional abstrata, e por último, a implementação física. A figura

2.1 mostra de forma esquemática o relacionamento existente entre os aspectos conceituais de uma aplicação e sua implementação.

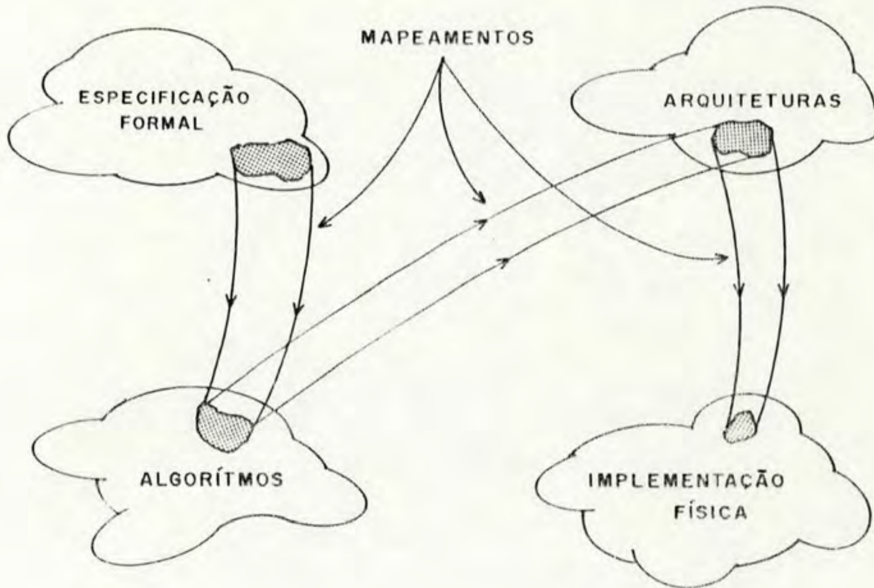



**figura 2.1 - Etapas no desenvolvimento de arquiteturas paralelas**

A especificação formal é empregada para definir formalmente os objetivos computacionais que pretende-se atingir. Sua função é precisar a forma e o conteúdo dos dados de entrada. Esta etapa fornece, ainda, as informações de como estes dados são operados para obter-se a saída desejada. Estas transformações são descritas através de informações lógicas, matemáticas, ou simbólicas, necessárias a produzir a saída especificada, a partir de um conjunto de entrada, para cada estado computacional existente.

Na etapa de algoritmo, descreve-se de forma algorítmica a solução resultante da análise computacional do problema. Este algoritmo não é único, é deve levar em

consideração o modelo de máquina em que será avaliado.

A etapa de arquitetura corresponde a um conjunto de restrições que devem ser considerados para a avaliação de um determinado algoritmo. São exemplos destas restrições, a performance desejada, a necessidade de comunicação entre processos, a possibilidade de implementação, enfim, condições que são impostas na obtenção da arquitetura.

Por último, a etapa de implementação, que preocupa-se com questões básicas de como a arquitetura obtida pode ser fisicamente realizada. Nesta etapa são abordados itens relacionados com a construtividade, como tecnologia a ser empregada, manufaturação, etc...

A relação aplicação-algoritmo-arquitetura exerce influência fundamental na exploração do paralelismo. Os principais fatores influenciados são: representação e descrição do paralelismo, estrutura de dados, mapeamento do algoritmo à arquitetura, esquemas para fluxo de dados e de controle, desempenho, entre outros.

As relações acima descritas são complexas, porém essenciais para o mapeamento de uma idéia abstrata em uma implementação física. Atualmente, o que falta, é um formalismo que represente estas relações de forma eficiente, sendo claro e facilmente manipulável. Hoje, grande parte destes problemas são solucionados de forma "ad hoc". Além das dificuldades encontradas na definição de algoritmos paralelos, e de seu mapeamento em arquiteturas, há o problema de implementação, pois na grande maioria das vezes, a programação de processadores paralelos é feita em linguagem de máquina, ou através de microprogramação.

Em função da importância vital dos problemas acima apresentados, muitos esforços estão sendo direcionados na criação de formalismo para o mapeamento de algoritmos à arquitetura, e na obtenção de linguagens que facilitem a exploração do paralelismo.

## 2.1 Obtenção de algoritmos paralelos

O maior problema encontrado na área do processamento paralelo é a tradução de uma aplicação para uma forma de avaliação paralela. Muitas vezes, algoritmos que demonstram, a uma primeira vista, características boas para o processamento paralelo acabam revelando-se ineficientes. Por exemplo, a necessidade de comunicação entre elementos processadores - em função do tipo de interconexão empregado - pode provocar um desempenho menor do que este mesmo algoritmo sendo executado em máquinas convencionais.

As principais dificuldades encontradas na criação de algoritmos paralelos advém de três fatores. Primeiro, a unidade de controle das máquinas paralelas em geral é uma unidade sequencial, o que implica em que procedimentos paralelos sejam organizados de uma forma sequencial. Segundo, a diferença conceitual de programação que existe entre os computadores convencionais e os paralelos. Por exemplo, em um processador do tipo matricial, o usuário (programador) deve ter sempre em mente que está operando sobre vários processadores, e a participação ou não de um processador em determinada operação deve ser feita de forma a mascará-lo. E por último, a identificação de pontos que possam ser paralelizados não é de fácil visualização, e às vezes, a simples troca de um índice, ou a inversão na ordem de uma parcela a ser calculada é suficiente para aumentar ou inviabilizar o desempenho da aplicação.

Segundo Voight [VOI 85] as principais abordagens a serem consideradas na obtenção de algoritmos paralelos em geral são: a análise da complexidade do problema, a manipulação da ordem de avaliação do problema, o aumento da quantidade de computações assíncronas, e o emprego de divisão e conquista. A análise de complexidade tem como objetivo a verificação da viabilidade do algoritmo a ser paralelizado

em função do paralelismo apresentado pela arquitetura. O conceito de alteração na ordem de computação pode ser visto como uma reordenação na sequência de operações visando um aumento na quantidade de operações que possam ser realizadas em paralelo. O emprego de computação assíncrona permite que não haja a necessidade de sincronizar processadores entre si, evitando que processadores que já efetuaram sua computação fiquem a espera de outros, (como exemplo, cita-se o que ocorre nas máquinas "data flow"). Por último, na divisão e conquista o particionamento do problema em sub-problemas menores de forma a serem tratados independentemente reduz a necessidade de sincronização e/ou comunicação entre elementos processadores. A divisão e conquista reduz, ainda, a complexidade de tratamento do problema sob o ponto de vista computacional.

O problema de considerar estes fatores, determinando sequências de tarefas a serem executadas em paralelo, e a melhor distribuição de dados e processos em elementos processadores, otimizando ou diminuindo a necessidade de comunicação ("data routing time"), é denominado de problema de mapeamento.

## 2.2 Mapeamento de algoritmos

A eficiência na avaliação de um algortimo computacional depende da arquitetura escolhida para sua execução [HON 79]. Em função desta dependência é que surge o problema do mapeamento de algoritmos às arquiteturas. A definição de estratégias para o problema do mapeamento não é recente, porém sofreu um grande impulso com o advento das arquiteturas sistólicas [KUN 82].

A estrutura de um algoritmo paralelo, idealmente, deve combinar com o tipo de arquitetura da máquina na qual será implementada. Para obter-se este mapeamento estrutural entre algoritmos paralelos e arquiteturas, Kung [KUN 80] propôs um esquema de classificação de algoritmos paralelos

correspondentes a uma classificação de arquiteturas paralelas.

Segundo Kung, um algoritmo paralelo pode ser considerado como um número de processos independentes que comunicam-se entre si gerenciados por algum tipo de controle. Em função desta comunicação, os algoritmos paralelos são caracterizados por sua granularidade, isto é, em função da quantidade de computação que um processo típico pode atingir antes de necessitar comunicar-se com outro para poder continuar processando. Por exemplo, se um processo possui uma alta taxa de comunicação com outro processo, a eficiência na avaliação será aumentada proporcionalmente em função das facilidades que a estrutura de interconexão da arquitetura empregada provê a estes processos. Pode-se concluir, então, que a necessidade de comunicação é um fator importante no mapeamento de algoritmos à arquiteturas.

O trabalho de Kung é extendido por Chiang [CHI 83] de forma a incluir, na eficiência de avaliação de um algoritmo paralelo, os fatores de movimentação de dados e organização de memória. Os principais itens abordados são: (a) o movimento de dados ocorre de forma regular ou irregular; (b) a memória empregada é local ou global; (c) havendo necessidade de comunicação entre processadores, isto ocorre de forma variante ou invariante.

No trabalho de Kung é proposto, ainda, uma classificação de arquiteturas em função do tipo de controle e de interconexão empregados. A classificação de Kung, para as arquiteturas paralelas, é mostrada de forma esquemática na figura 2.2.

UFRGS
BIBLIOTECA
CPD/PGCC

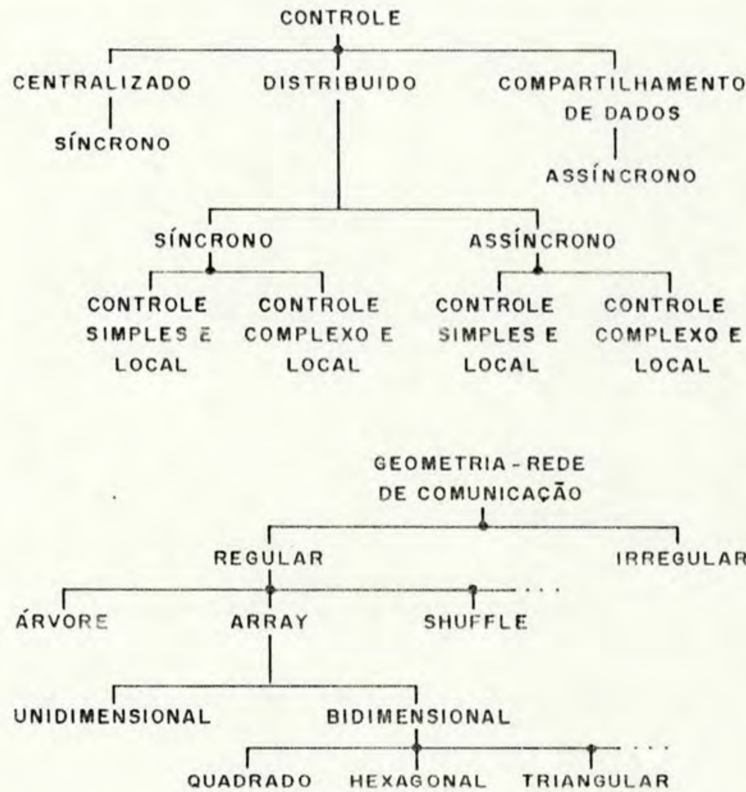



**figura 2.2 - Esquema para classificação de arquiteturas paralelas**

Com o emprego dos esquemas de classificação acima, pode-se caracterizar, de forma abrangente, classes de algoritmos que combinam com classes de arquiteturas. Por exemplo, a figura 2.3 relaciona algumas características a serem consideradas no mapeamento de algoritmos às arquiteturas.

Apesar dos esforços desenvolvidos, o problema do mapeamento não está de todo solucionado. Falta, ainda, uma metodologia formal que reúna os fatores de comunicação, organização e movimento de dados, com as principais características arquiteturais de forma eficiente. A seguir apresentar-se-á um breve apanhado de trabalhos desenvolvidos nesta área.

| TIPO DE ARQUITETURA | SIMD | MIMD | SISTÓLICO |
|---|---|---|---|
| GRANULARIDADE | PEQUENA | GRANDE | PEQUENA |
| CONTROLE | CENTRALIZADO SÍNCRONO | ASSÍNCRONO COMPARTILHAMENTO DE DADOS | DISTRIBUIDO SÍNCRONO |
| REDE DE INTERCONEXÃO | REGULAR | IRREGULAR | REGULAR |
| ORGANIZAÇÃO DE MEMÓRIA | LOCAL PEQUENA/GRANDE | GLOBAL LOCAL GRANDE/PEQUENA | LOCAL PEQUENA |

**figura 2.3 - Características de algoritmos e arquiteturas**

Flander [FLA 82], afirma que o desempenho no uso de processadores matriciais está vinculada a maneira pela qual os dados são organizados: propõe em seu trabalho uma estrutura de organização de dados para minimizar o tempo gasto no roteamento de dados. Bokhari [BOK 81], baseado na teoria de grafos, propôs um algoritmo heurístico para mapeamento de problemas. Aggarwal e Lee [AGG 87], também empregando teoria de grafos, abordam a necessidade de comunicação em função da dependência de tarefas. Estratégias de mapeamento abordando dependência entre variáveis como forma de detectar pontos que possibilitem computação em paralelo são fornecidos em [KUN 88], [MOL 83] e [LIN 85].

## 2.3 Avaliação de algoritmos paralelos

Um fator importante a ser considerado na utilização de algoritmos paralelos é a avaliação de seu desempenho e eficiência. A importância desta avaliação é permitir uma análise da qualidade do mapeamento do algoritmo à arquitetura. Outro fator a ser considerado é a determinação de a partir de que complexidade computacional torna-se viável a sua execução em uma arquitetura paralela. Aqui, novamente, a obtenção de informações sistemáticas não é uma tarefa fácil, não existindo meios adequados de avaliação. O maior problema encontrado é que tipo de padrão de medida empregar, já que um determinado padrão pode quantizar aspectos particulares de um fator negligenciando outros.

Uma ferramenta comumente empregada na avaliação de algoritmos é a análise de complexidade. O emprego de análise de complexidade, porém, segundo Voight [VOI 85], não demonstra-se eficiente por não considerar dois aspectos básicos: (a) as máquinas paralelas podem suportar certas computações adicionais sem queda de performance ao explorar organizações adequadas de dados, e (b) existem "overheads" de comunicação e sincronização. Deve-se, então, criar outros dispositivos para avaliar eficientemente o desempenho real de tais algoritmos.

Os estudos sobre desempenho, hoje em dia, estão sendo feitos de uma maneira muito global, baseando-se na escolha de um algoritmo e adaptando-o a uma arquitetura. A partir deste ponto são desenvolvidos análises para determinação da performance. Na prática, de uma forma simples, a performance de um arquitetura paralela pode ser medida em termos de um "speed up" alcançado sobre uma máquina sequencial [OFF 85].

## 2.4 Linguagens voltadas a exploração de paralelismo

A criação de linguagens para expressão de paralelismo não é uma tarefa trivial. Um dos maiores problemas é a determinação automática de pontos a serem paralelizados, que é muito complexa, sendo que os melhores resultados obtidos até hoje foram com compiladores vetorizadores. Os quais mesmo assim, geram código em duas etapas: uma primeira automaticamente, e uma segunda iterativamente com o usuário. Outro ponto desfavorável à criação das linguagens paralelas é o fato que muitas vezes as estruturas a serem criadas para explorar o paralelismo estão fortemente vinculadas à máquina em que serão executadas e a suas aplicações. Uma linguagem paralela deve provêr, ainda, meios que facilitem a manipulação de dados entre os elementos processadores (comunicação). Devido a estes fatores, as linguagens paralelas até hoje desenvolvidas são vinculadas a uma determinada máquina, não existindo uma linguagem que seja universal.

Buscando suprir esta lacuna, Perrot [PER 79] propôs uma linguagem para implementar-se programas em máquinas do tipo matricial. Esta linguagem, porém, devido a sua generalidade, apresenta como grande desvantagem uma exploração ineficiente do paralelismo.

No desenvolvimento de linguagens paralelas, e na expressão do paralelismo, existem duas abordagens a serem consideradas [PER 79]. Na primeira abordagem o usuário escreve o programa de forma convencional, ou seja, de uma maneira sequencial, sendo o compilador da linguagem o responsável pela detecção e exploração do paralelismo no programa. Esta filosofia de expressão de paralelismo foi empregado na criação das linguagens IVTRAN (ILLIAC IV), e CFT-Cray FORTRAN Translator (CRAY), que exploravam o paralelismo em laços do tipo DO. Esta abordagem apresenta como

principal desvantagem o fato de que a extração do paralelismo é limitada, e que a geração de código não é otimizada.

O método empregado pela segunda abordagem é considerar no desenvolvimento da linguagem a natureza paralela da máquina, fornecendo a linguagem estruturas que possibilitam explorar a forma de armazenamento e a organização empregada pela máquina. São exemplos desta abordagem a linguagem GLYPNIR [LAW 75], ALGOL-like desenvolvida para o ILLIAC IV, e a linguagem CFD [STE 75], FORTRAN-like, também desenvolvida para o ILLIAC IV. Esta segunda abordagem apresenta como desvantagem uma maior complexidade de programação.

Como resultados mais recentes de pesquisas na obtenção de linguagens paralelas pode-se citar as extensões propostas para a linguagem ADA [CLI 85], e para a linguagem C [KUE 85]. Um exemplo de linguagem desenvolvida especificamente para uma máquina, explorando as características de sua arquitetura, é a linguagem GAL [NCR 85a] (GAPP Algorithm Language), C-like, desenvolvida pela NCR para o chip GAPP (Geometric arithmetic Parallel processor).

## 2.5 Restrições a algoritmos para arquiteturas SIMD

Como já mencionado nos itens anteriores, o desempenho e a eficiência na avaliação de um algoritmo depende da arquitetura selecionada para sua execução. Fatores como a organização de dados em memória e comunicação entre processadores devem ser cuidadosamente considerados. Estas estruturas devem combinar o mais possível com a topologia empregada na comunicação processador-memória da arquitetura.

Particularmente, na obtenção de algoritmos para a classe de máquinas matriciais os principais fatores que devem ser considerados são: a complexidade aritmética das

operações a serem executadas, a necessidade de comunicação entre elementos processadores, o tamanho e organização da memória, e o número de processadores. Estes fatores estão relacionados com a implementação física destas arquiteturas.

Os processadores matriciais, na maioria das vezes, são compostos por elementos processadores bastante simples: uma ULA com as operações lógicas básicas, e com as operações de adição e subtração organizadas de forma serial por bit. Pode-se facilmente concluir que a execução de operações mais complexas, como a multiplicação e a divisão, oneram significativamente o tempo de computação.

Outra característica construtiva dos processadores matriciais, em função de custo e viabilidade de implementação física, é o emprego de redes de interconexão que comunicam um processador a seus vizinhos mais imediatos. Como consequência, deve-se evitar aplicações que exijam comunicação com processadores distantes e/ou de forma variante no tempo.

A organização da memória, local ou global, aliada ao tipo de interconexão existente entre os processadores, representa um fator fundamental na manipulação de dados. O emprego de memória local reduz a necessidade de acesso à uma memória global, eliminando assim um gargalo no sistema. A desvantagem do uso de memória local está na necessidade de criar meios otimizados para recuperação e carga de dados.

Finalmente, o número de processadores disponíveis na matriz de processadores, que pode em função do problema, determinar que este seja particionado em subproblemas menores permitindo que seja processado na matriz. Esta subdivisão causa um "overhead" de controle e de comunicação muito grande, pois os dados devem ser carregados na matriz, processados, e recuperados, até que toda a computação do problema seja efetivada.

A escolha de uma aplicação (problema) e de sua implementação deve ser feita a partir destas considerações básicas. Em função destas considerações, é importante, ainda, levar em conta que o tempo de execução de um algoritmo em processadores matriciais é constituído de três parcelas: tempo de execução (processamento propriamente dito), tempo de comunicação interprocessadores, e tempo de E/S dos dados na memória do processador matricial, os quais devem ser sempre minimizados.

U F R G S
BIBLIOTECA
CPD/PGCC

## 3 PROCESSAMENTO DE IMAGENS: UM EXEMPLO DE APLICAÇAO

As primeiras pesquisas relacionadas com a área de processamento digital de imagens remontam da década de 50 a partir de trabalhos na área de reconhecimento de padrões, onde o principal objetivo era a associação de uma imagem com uma série de padrões previamente definidos.

A década de 60 foi marcada por um crescimento de interesse e de esforços na área de processamento digital de sinais, e em especial ao tratamento de imagens, motivadas, principalmente, em função do programa espacial norte americano. Esta fase inicial caracterizou-se pelo desenvolvimento e pesquisas de técnicas para o processamento de imagens. São ferramentas típicas resultantes destes esforços as operações de filtragem, técnicas de codificação de imagens, e a transformada rápida de Fourier (FFT).

O emprego das técnicas de processamento de sinais, em particular, da transformada rápida de Fourier popularizou as aplicações de tratamento de imagens. Concomitantemente com o crescimento de áreas de aplicação começaram a surgir melhorias consideráveis na velocidade, no tamanho, versatilidade e no custo de computadores digitais, bem como de seus periféricos, isto em parte como uma conseqüência natural dos avanços significativos ocorridos com as tecnologias de LSI e VLSI. Áreas como medicina, física nuclear, sensoramento remoto, entre outras, são exemplos de aplicações que empregam o processamento digital de imagens para a obtenção de informações relevantes.

Devido a grande quantidade de informações a serem processadas no tratamento de imagens, em especial, nos sistemas em tempo real, surgiu a necessidade de desenvolver-se sistemas, ou componentes, que obtivessem uma melhor performance nesta avaliação que as tradicionais máquinas de Von Neumann até então empregadas. Como exemplo pode-se citar máquinas matriciais e unidade aritméticas pipelíniza-

das como o ILLIAC, STARAN, DAP, e mais recentemente, MPP, GAPP e GRID.

## 3.1 Caracterização do problema processamento de imagens

A principal finalidade de um sistema de processamento de imagens é, a partir de uma imagem obtida por algum processo de digitalização, extrair informações através de transformações sucessivas na imagem amostrada. O objetivo de transformar-se uma imagem é de salientar características desejáveis, obtendo apenas informações relevantes para uma dada aplicação, reduzindo assim o volume de dados a ser processado.

Em Offen [OFF 85] é apresentado uma taxonomia para sistemas de processamento de imagens baseado no tipo de processamento requerido, sendo esquematizado em três (3) níveis hierárquicos, a saber: dados, informação e conhecimento. No nível de dados, hierarquicamente o inferior, a imagem é tratada como se fosse uma amostra de uma forma de onda em um instante "t" no tempo. Esta amostra, porém, é organizada em uma forma bidimensional. Neste nível a imagem é considerada apenas como um conjunto de dados ao invés de uma representação de alguma cena, não havendo referências quanto a estruturas, ou objetos que formam a imagem. São exemplos de operações que compõem este nível as operações de filtragens e codificação de imagens como o DPCM (Differential Pulse Code Modulation).

O nível de informação tem como principal objetivo a manipulação e a extração de informações a partir da imagem elaborada pelo nível inferior (dados). São exemplos típicos de informações obtidas neste nível a definição de contornos, formas de objetos a partir do emprego de operadores como Marr-Hildret [OFF 85] e transformada de Hough [DUD 72].

O nível hierárquico mais alto, o de conhecimento,

tem como função interpretar as informações empregando um conhecimento sobre o domínio (imagem). Exemplos típicos para este nível são operações que envolvem o reconhecimento de objetos, modelagem tridimensional, e a descrição de cenas.

O esquema hierárquico apresentado acima para classificação de sistemas de processamento de imagens não fornece uma forma precisa para classificação de arquiteturas e algoritmos por ser difícil, em alguns casos, de especificar em qual nível estes operam. A vantagem do emprego deste esquema de classificação está em, primeiro, uma grande parte dos algoritmos podem ser corretamente classificados, e segundo, existe uma correlação entre os níveis hierárquicos e o conceito de cone de processamento ("processing cone") [HWA 84].

O conceito de cone de processamento, figura 3.1, está relacionado com uma analogia frequentemente empregada entre um cone e o rápido aumento de necessidade computacional a nível de flexibilidade e controle. Por exemplo, na base do cone estão os processadores massissamente paralelos, que são relativamente inflexíveis na natureza das operações que executam, sendo ideais para o processamento de grande conjunto de dados que são avaliados de uma mesma forma. Subindo-se na hierarquia, a quantidade de paralelismo e dados a serem manipulados são reduzidos, e a necessidade de maior controle e flexibilidade de operações são aumentadas.

Pode-se observar então uma relação entre o cone de processamento e a taxonomia de hierarquia concluindo-se que os algoritmos que são executados no nível hierárquico de dados são possíveis candidatos a uma implementação com processadores massissamente paralelos explorando o paralelismo inerente a este nível.

figura 3.1 - Cone de processamento

## 3.2 Arquiteturas paralelas para processamento de imagens

As operações tipicamente efetuadas na transformação de uma imagem são a computação de uma função não negativa, geralmente números inteiros, correspondendo a níveis de cinza, ou níveis de intensidade, de todos os elementos que compõem a imagem.

Considerando que a imagem digitalizada seja representada por uma matriz I(x,y); se o valor (nível de cinza) de um pixel no ponto (x,y) é indicado por P, o valor obtido após a aplicação da função de transformação é P', tal que, P'=T(V(P)), onde T é a função de transformação e V(P) são valores dos pixels "vizinhos" a P.

A "vizinhança" de P, ou "vizinhos" de P, são definidos como: dado um ponto P(x,y) da imagem, a "vizi-

nhança-4" de P é o conjunto dos quatro pontos (x-1,y), (x+1,y), (x,y-1) e (x,y+1), e a "vizinhança-8", a união entre os pontos da "vizinhança-4" e os pontos (x-1,y-1), (x+1,y+1), (x-1,y+1) e (x+1,y-1). A distância entre um ponto P e seus "vizinhos-4" é a unidade (1), e a distância entre um ponto e seus "vizinhos-8", que não são "vizinhos-4", é dois (2).

A função de transformação T, por dever ser avaliada para todos os pontos que compõem a imagem I apresenta um alto grau de paralelismo. Idealmente este paralelismo pode ser visto bidimensionalmente no espaço, associando-se a cada ponto (pixel) um elemento de processamento capaz de comunicar-se com os processadores vizinhos para avaliar a função de transformação T. Estas necessidades vem de encontro com as características apresentadas pelas arquiteturas do tipo SIMD.

Mesmo com o avanço das técnicas de microeletrônica, em especial VLSI, torna-se difícil projetar-se componentes que contenham matrizes de processadores em tamanhos típicos de imagens, como 256x256, ou 512x512. A solução adotada é possibilitar que os processadores sejam os mais simples possíveis, e que a imagem possa ser tratada em matrizes menores através da partição da imagem em sub-imagens.

Estas limitações de ordem prática, na implementação de tais sistemas, levam a necessidade de elaborar-se algoritmos paralelos que respeitem estas restrições. Na elaboração de tais algoritmos são considerados, conforme apresentado no capítulo 2, os fatores de comunicação entre processadores e com o meio externo, a organização interna dos dados, e a simplicidade em operações aritméticas e lógicas. A seção a seguir apresenta alguns dos algoritmos mais comumente empregados no tratamento de imagens, no nível hierárquico de dados.

UFRGS BIBLIOTECA

## 3.3 Algoritmos para processamento de imagens

A finalidade principal de um sistema de processamento de imagens é a extração de informações a partir de uma imagem. Esta extração é realizada a partir de transformações executadas na imagem original possibilitando que determinadas características sejam mais enfatizadas que outras. São exemplos destas transformações a correção de distorções (restauração), e o realce (enfatização de características), as quais empregam técnicas de filtragens. A diferença entre estas operações são apenas de objetivos e enfoques.

As transformações geralmente realizadas sobre imagens são de dois tipos: locais e pontuais. Na transformação pontual o novo valor para o nível de cinza de um ponto depende exclusivamente do nível de cinza do ponto considerado. A transformação local, por outro lado, considera para a avaliação do novo nível de cinza a influência (nível de cinza) dos pontos vizinhos.

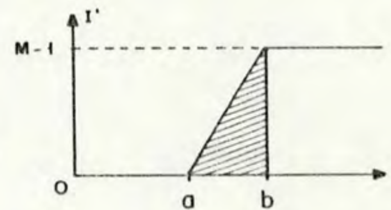
### 3.3.1 Mapeamento de níveis de cinza

O mapeamento de níveis de cinza é uma operação pontual, cujo principal objetivo é salientar pontos na imagem que possuam determinadas características comuns. Este tipo de operação também é denominada de "thresholding".

Este tipo de transformação opera diretamente nos níveis de cinza que compõem a imagem executando um mapeamento de valores de um intervalo [0,N-1] para um intervalo [0,M-1], onde o intervalo [0,N-1] representa os valores originais dos níveis de cinza que compõem a imagem, e o intervalo [0,M-1] um novo conjunto de valores para estes níveis de cinza. Este mapeamento pode ser obtido através de uma função, de uma expressão aritmética, ou com o emprego

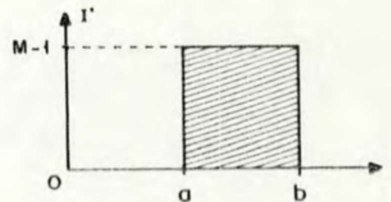
de tabelas. Por exemplo, pode-se obter a partir de uma imagem com 256 níves de cinza (N=256, intervalo [0,255]), uma nova imagem com 2 níveis (N=2, intervalo[0,1]) empregando-se a função dada na figura 3.2c. Neste caso, os valores assumidos pela nova imagem serão sempre zero enquanto o valor do nível de cinza da imagem original for menor do que um determinado nível "a" do intervalo [0,255]; o nível de cinza da nova imagem será um (1) caso contrário.

$$I'(x,y) = \begin{cases} 0, & \text{se } I(x,y)<a \\ M-1, & \text{se } I(x,y)>b \\ I(x,y), & \text{caso contrário} \end{cases}$$



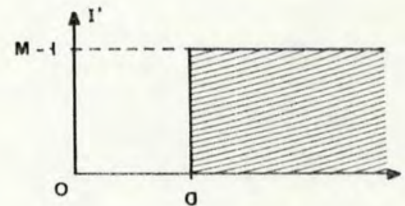

(a)

$$I'(x.y) = \begin{cases} 0, & \text{se } I(x,y)<a \\ 0, & \text{se } I(x,y)>b \\ M-1, & \text{caso contrário} \end{cases}$$




(b)

$$I'(x,y) = \begin{cases} 0, & \text{se } I(x,y)<a \\ M-1, & \text{se } I(x,y)>= a \end{cases}$$




(c)

**Figura 3.2 - Exemplos de funções para mapeamento de níveis de cinza**

A escolha adequada de função de mapeamento implica na necessidade dos elementos processadores executarem operações de comparação entre limites, o que é facilmente obtido com operações lógicas e aritméticas elementares. Observa-se, ainda, que em função da transformação, via mapeamento de níveis de cinza, ser do tipo pontual o "overhead" necessário para a comunicação entre processadores é nulo.

### 3.3.2 Filtragem

O objetivo principal da maioria das operações de filtragens é atuar sobre as intensidades de nível de cinza que compõem a imagem de forma a realçar, ou suavizar, determinadas características. Por exemplo o emprego de filtros passa-baixas acarreta uma suavização ("smoothing") da imagem obtendo-se uma atenuação nas altas frequências, usualmente relacionadas com ruídos no processo de digitalização da imagem. Por outro lado, a utilização de filtros passa-altas fornece uma atenuação em baixas frequências sendo empregado para realçar bordas e contornos.

O embasamento matemático das operações de filtragens sobre imagens é a Transformada de Fourier. Devido ao fato de não ser a ênfase principal deste trabalho um estudo sobre Transformadas de Fourier e de técnicas de filtragem, não serão apresentados os conceitos relacionados a este tema, atendo-se apenas aos seus resultados como ferramentas. Pode-se, porém, enumerar uma série de publicações sobre o assunto como [RAB 75], [MAS 84] e [PRA 78].

As operações de filtragens são obtidas através de uma convolução entre uma imagem I e uma imagem W, sendo que a imagem W, constituída por uma matriz de pesos, é o filtro. Na realidade o resultado da convolução da imagem I com a imagem W é uma média ponderada dos pontos de I, onde os pesos são fornecidos pela imagem W. A matriz W (imagem)

também é denominada de máscara, e possui uma dimensão (2n+1)x(2n+1). A escolha do valor "n", devido a necessidades computacionais, recai em valores pequenos (no máximo 3) [MAS 84]. Tipicamente "n" é igual a um (1), o que fornece uma matriz W de peso de dimensão 3.

Os coeficientes (pesos) que compõem a matriz W, e a sua dimensão, determinam as características do filtro. Por exemplo, uma operação de suavização pode ser obtida com a aplicação, para cada pixel que compõem a imagem, da máscara fornecida na figura 3.3. O aumento do fator "a" na matriz possui o efeito de reduzir o grau de borramento produzido.

$$W(k,l) = \begin{matrix} 1 & 1 & 1 \\ 1 & a & 1 \\ 1 & 1 & 1 \end{matrix}$$

**Figura 3.3 - Máscara para operação de suavização**

Uma característica apresentada pelas máscaras é a sua simetria em relação ao ponto central da matriz. Esta simetria auxilia a avaliação da convolução por fornecer na análise no domínio frequência uma parte imaginária nula [MAS 84].

A figura 3.4 [MAS 84] fornece máscaras empregadas na detecção de bordas em vários sentidos. A utilização destas máscaras para a detecção de bordas implica, porém, em um "overhead" computacional grande, já que para detectar-se bordas em todos as direções é necessário avaliar a imagem várias vezes com máscaras diferentes. No entanto, existem aproximações que permitem a avaliação de bordas independentemente de sua direção. O conjunto de máscaras fornecidos na figura 3.5 [MAS 84] implementam uma aproximação do operador Laplaciano aplicado à função bidimensional que é a imagem. A avaliação do Laplaciano da imagem permite

o realce de bordas independente da direção.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | 1 | 1 | 1 |
| Norte | 1 | -2 | 1 | Nordeste | -1 | -2 | 1 |
| | -1 | -1 | -1 | | -1 | -1 | 1 |
| | | (a) | | | | (b) | |
| | -1 | 1 | 1 | | -1 | -1 | 1 |
| Leste | -1 | -2 | 1 | Sudeste | -1 | -2 | 1 |
| | -1 | 1 | 1 | | 1 | 1 | 1 |
| | | (c) | | | | (d) | |
| | -1 | -1 | -1 | | 1 | -1 | -1 |
| Sul | 1 | -2 | 1 | Sudoeste | 1 | -2 | -1 |
| | 1 | 1 | 1 | | 1 | 1 | 1 |
| | | (e) | | | | (f) | |
| | 1 | 1 | -1 | | 1 | 1 | 1 |
| Oeste | 1 | -2 | -1 | Noroeste | 1 | -2 | -1 |
| | 1 | 1 | -1 | | 1 | -1 | -1 |
| | | (g) | | | | (h) | |

Figura 3.4 - Máscara para detecção de bordas

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | -1 | -1 | -1 | 1 | -2 | 1 |
| 1 | -4 | 1 | -1 | 8 | -1 | -2 | 4 | -2 |
| 0 | 1 | 0 | -1 | -1 | -1 | 1 | -2 | 1 |
| | (a) | | | (b) | | | (c) | |

Figura 3.5 - Máscara para operação de Laplaciano

A aproximação realizada pelas máscaras da figura 3.5 pode ser observada partindo-se da definição do Laplaciano:

$$\nabla^2 f = \nabla^2_x f + \nabla^2_y f$$

Se as derivadas direcionais são aproximadas pelas diferenças

$$\nabla_x I(x+1,y) = I(x+1,y) - I(x,y)$$

$$\nabla_y I(x,y+1) = I(x,y+1) - I(x,y)$$

e

$$\nabla_x I(x,y) = I(x,y) - I(x-1,y)$$

$$\nabla_y I(x,y) = I(x,y) - I(x,y-1)$$

As derivadas segundas ficam:

$$\nabla^2_x I(x,y) = \nabla_x I(x+1,y) - \nabla_x I(x,y)$$

$$= I(x+1,y) - 2.I(x,y) + I(x-1,y)$$

$$\nabla^2_y I(x,y) = \nabla_x I(x,y+1) - \nabla_x I(x,y)$$

$$= I(x,y+1) - 2.I(x,y) + I(x,y-1)$$

Consequentemente, o Laplaciano no ponto I(x,y) é dado por:

$$\nabla I(x,y) = -4.I(x,y) + I(x-1,y) + I(x+1,y) + I(x,y-1) + I(x,y+1)$$

o que corresponde a avaliação da máscara da figura 3.5a.

Outros operadores comumente empregados para a detecção de bordas, que independem da direção da borda, são os operadores de gradiente de Roberts e de Sobel [MAS 84],

[NCR 85b]. Estes operadores atuam combinando a avaliação das diferenças de intensidade no nível de cinza que compoẽm a imagem em duas direções ortogonais ao ponto sob cálculo.

O operador gradiente de Sobel é preferido em relação ao de Roberts, devido ao fato que o operador de Roberts apresenta uma assimetria, que em função da direção da borda, salienta mais certas bordas que outras, mesmo possuindo igual magnitude. O operador gradiente de Sobel é dado por:

$$G_s = (X_s^2 + Y_s^2)^{1/2} \qquad (I)$$

onde,

$$X_s = [I(x-1,y+1) + 2.I(x,y+1) + I(x+1,y+1)] - [I(x-1,y-1) + 2.I(x,y-1) + I(x+1,y-1)] \qquad (II)$$

e,

$$Y_s = [I(x-1,y-1) + 2.I(x-1,y) + I(x-1,y+1)] - [I(x+1,y-1) + 2.I(x+1,y) + I(x+1,y+1)] \qquad (III)$$

A obtenção do gradiente de Sobel é realizada com a filtragem da imagem com uma matriz de peso (máscara) na direção X, e na direção Y, avaliando $X_s$ e $Y_s$, respectivamente. A figura 3.6 fornece as máscaras empregadas para o cômputo de $X_s$ e $Y_s$. Convém salientar que convenciona-se em processamento de imagens a direção X como sendo o número de linhas que compõem a imagem, e a direção Y o número de colunas.

$$\text{direcao X} = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad \text{direcao Y} = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

**figura 3.6 - Máscara para operação de Sobel**

O gradiente de Sobel pode, se necessário, fornecer uma estimativa da direção da borda detectada computando-se arctan($Y_s/X_s$).

A avaliação do gradiente de Sobel ($G_S$) apresenta uma alto custo computacional devido a necessidade de operações de elevar ao quadrado e de extrair a raiz quadrada. Os processadores SIMD devido a sua simplicidade apresentam um "overhead" maior na avaliação destas operações, principalmente se for do tipo serial por bit.

Como apresentado no capítulo 2, um dos fatores que deve ser considerado é a complexidade aritmética das operações a serem realizadas. Uma forma de contornar, ou reduzir, esta limitação é o emprego de aproximações. A aproximação empregada na avaliação da expressão (I), que fornece o gradiente de Sobel, e'[NCR 85b]:

$$G_S = MAX(|X_s|,|Y_s|) + 1/2\ MIN(|X_s|,|Y_s|) \qquad (IV)$$

que pode ser obtida de forma relativamente simples empregando apenas operações lógicas para determinar as parcelas MAX e MIN. Observa-se ainda que a divisão por 2 pode ser facilmente obtida com a utilização de um deslocamento (shift) à direita.

### 3.4 Implementação do algoritmo Sobel em arquiteturas SIMD

Com o objetivo de ilustrar a utilização dos algoritmos apresentados na seção anterior em arquiteturas do tipo SIMD, salientando-se as implicações na definição destas arquiteturas, para uma eficiente implementação, empregar-se-á o algoritmo de avaliação do gradiente de Sobel. A arquitetura alvo a ser considerada será uma arquitetura SIMD serial por bit, como a descrita no capítulo 1, seção 1.2.1.

A avaliação do gradiente de Sobel é realizado

computando-se a expressão fornecida em (IV). Definindo-se uma região vizinha a um pixel P de coordenada (x,y) como sendo uma matriz 3x3, onde o elemento $m_{22}$ representa este pixel P, a partir de (II) e (III) avalia-se $Y_s$ e $X_s$ como:

$$X_s = (m_{13} + 2*m_{23} + m_{33}) - (m_{11} + 2*m_{21} + m_{31}) \qquad (V)$$

$$Y_s = (m_{11} + 2*m_{12} + m_{13}) - (m_{31} + 2*m_{32} + m_{33})$$

Como esta avaliação é necessária a todos os pixels P que compoẽm a imagem, a exploração deste paralelismo pode ser obtido empregando uma matriz de processadores. Esta organização pode implicar em que a cada pixel da imagem esteja associado um processador. As variáveis necessárias para a avaliação de $G_S$, bem como a intensidade do nível de cinza do pixel P, podem ser armazenadas em uma memória local pertencente a cada processador. Devido a necessidade de ao encontrar-se $G_S$ considerar os valores dos pixels imediatamente vizinhos, a interconexão a ser empregada na matriz de processadores deve prover comunicação entre os 8 processadores vizinhos e o processador do pixel considerado (figura 3.7a). Como a implementação física de tal interconexão começa a tornar-se inviável, em função da grande quantidade de linhas, uma interconexão do tipo mesh (figura 3.7b) pode ser empregada.

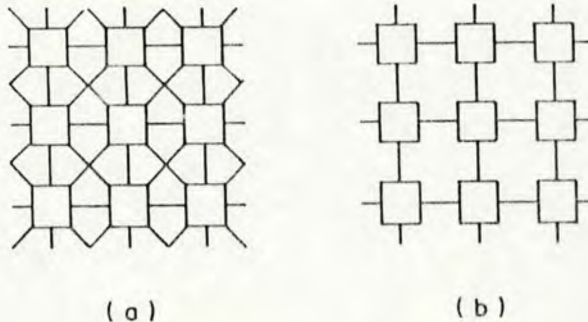



**figura 3.7 - Opções de interconexão**

Pode-se observar aqui a inclusão de um "overhead" adicional na comunicação ao optar-se pela rede de 3.7b invés de 3.7a, pois a comunicação entre o elemento $P_{11}$ ao $P_{22}$ deve ser realizado via $P_{12}$, ou $P_{21}$.

Analisando a equação (IV) verifica-se que é necessário, para cada pixel, a avaliação de $X_s$, $Y_s$, $MAX(|X_s|,|Y_s|)$, $MIN(|X_s|,|Y_s|)$. Estas variáveis sendo armazenadas na memória local de cada processador, em um mesmo endereço, definindo planos de memória, permitem que todas as computações ocorram relacionadas a um mesmo endereço fonte e destino, possibilitando que uma única instrução possa atuar sobre todos elementos processadores. Esta organização de memória pode ser vista na figura 3.8.

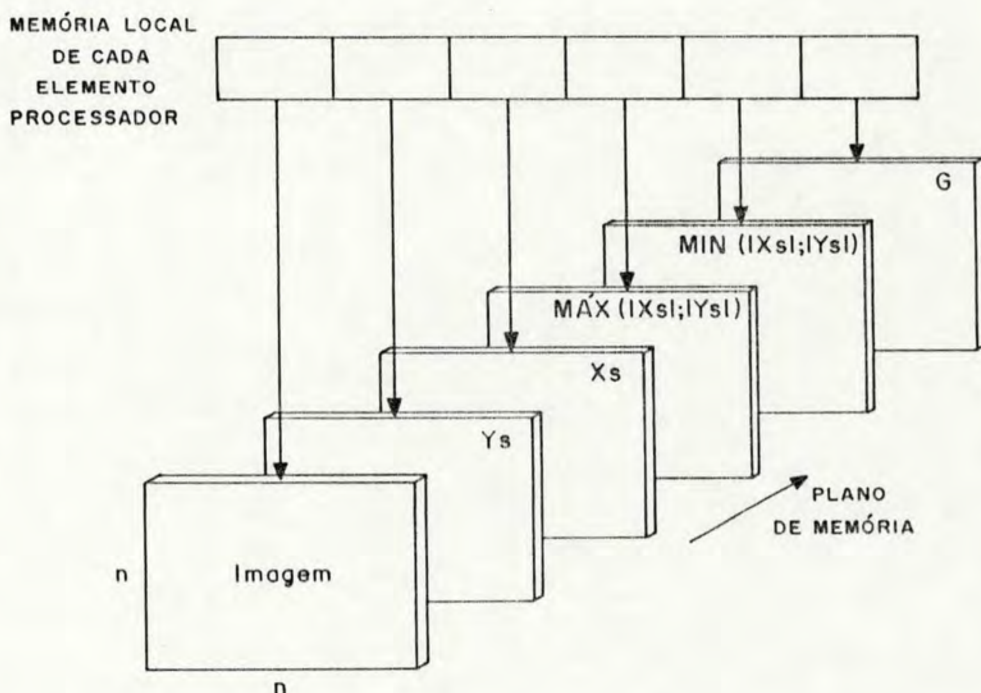



figura 3.8 - Organização de memória

A forma de conexão (mesh), e a maneira pela qual os dados estão organizados, podem ser aproveitados para aumentar o desempenho na avaliação de $G_S$ ao reescrever-se a equação (V) da seguinte forma:

$$X_s = [(m_{13}+m_{23}) + (m_{23}+m_{33})] - [(m_{11}+m_{21}) + (m_{21}+m_{31})]$$
$$Y_s = [(m_{11}+m_{12}) + (m_{12}+m_{13})] - [(m_{31}+m_{32}) + (m_{32}+m_{33})] \qquad \text{(VI)}$$

A exploração de paralelismo na obtenção de $G_S$ pode ser melhor visualizada analisando-se, a partir de um exemplo, os passos necessários para a avaliação de $Y_S$.

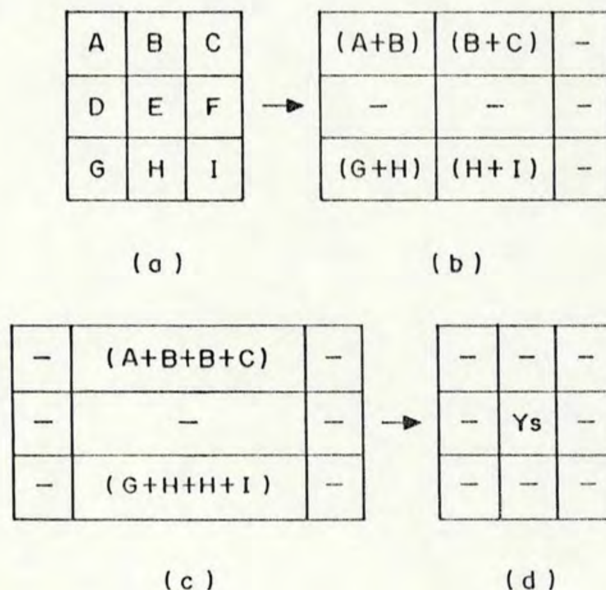



figura 3.9 - Avaliação de $Y_S$

Partindo-se da condição exemplo (figura 3.9a) deseja-se calcular o gradiente $Y_S$ do ponto "E", $G_S$(E). A equação (VI) pode ser então reescrita como:

UFRGS
BIBLIOTECA
CPD

$$Y_s = [(A+B) + (B+C)] - [(G+H) + (H+I)] \qquad (VII)$$

analisando-se a equação (VII) observa-se que as parcelas soma de $Y_s$ podem ser obtidas somando-se os conteúdos dos processadores com os conteúdos dos processadores vizinhos imediatamente à direita (figura 3.9b).

A obtenção da parcela soma entre colchetes é realizada somando-se para cada processador, a soma parcial obtida no passo anterior com a soma parcial do vizinho imediatamente à esquerda (figura 3.9c). Para obter-se $Y_s$ (figura 3.9d) subtrai-se as parcelas entre colchetes.

Observa-se, ainda, que a conexão do tipo mesh facilita a movimentação de dados entre os processadores nas direções norte-sul, leste-oeste. Pode-se facilmente imaginar que este procedimento, sendo realizado em todos elementos processadores, permite a avaliação em paralelo de $Y_s$ para todos pixels, pois a cada pixel está associado um processador. O gradiente $X_s$ é obtido de forma análoga.

A partir da determinação de $Y_s$ e $X_s$ o processamento se restringe a própria memória local do processador, o que é uma característica interessante, pois novamente todos processadores podem efetuar operações em paralelo. A determinação de $MAX(|X_s|,|Y_s|)$ e $MIN(|X_s|,|Y_s|)$ pode ser obtida diminuindo-se, em módulo, $X_s$ de $Y_s$ e verificando-se a existência ou não de "borrow". Em função do "borrow" determina-se as variáveis MAX e MIN.

Após avaliados $MIN(|X_s|,|Y_s|)$ e $MAX(|X_s|,|Y_s|)$ calcula-se o gradiente G pela expressão dada em (IV). A divisão por 2 é facilmente obtida executando-se uma operação de deslocamento à direita no conteúdo da palavra que armazena $MIN(|X_s|,|Y_s|)$. Obtém-se, então o gradiente GS somando-se os conteúdos das palavras MAX e MIN.

O código que implementa este algoritmo é apresentado no capítulo 6 deste trabalho.

### 3.5 Considerações sobre aplicações em máquinas SIMD

O emprego de arquiteturas especiais está fortemente relacionada com uma determinada aplicação. A definição da arquitetura a ser utilizada é função do tipo de paralelismo inerente apresentado pela aplicação. Para obter-se um ganho real no desempenho de uma arquitetura paralela, se comparado com o de uma convencional, deve-se, além de explorar adequadamente este paralelismo inerente, considerar os compromissos enumerados no capítulo 2.

Para as arquiteturas tipo SIMD os principais fatores a serem considerados são: comunicação interprocessadores, comunicação com o meio externo, organização dos dados em memória, e complexidade aritmética das operações a serem realizadas. Estes fatores são derivados de uma análise mais pragmática da figura 2.3, que salienta as características de granularidade, controle, rede de interconexão e organização de memória que um algoritmo deve possuir para se adequar a um determinado tipo de arquitetura.

Inicialmente, a granularidade, isto é, a quantidade de computação que pode ser executada por um processador antes de ser necessário efetuar uma comunicação com outro processador, se traduz nos fatores de comunicação interprocessadores e comunicação com o meio externo. É desejável, então, a redução de ambos, pois a não consideração destes pode acarretar uma degradação no tempo total de avaliação em decorrência do "overhead" causado pelo fluxo de dados.

A rede de interconexão, por sua vez, define quais os meios físicos existentes para ser efetuada a comunicação interprocessadores. A determinação da topologia da rede de interconexão influi de forma decisiva no tempo gasto nesta. Por questões de implementação física é desejável empregar-se uma rede de interconexão simples, o que se traduz em ser

regular e limitada. Por limitada entende-se a característica de um elemento processador possuir interconexão direta com apenas alguns elementos processadores que compõem o sistema. Como exemplo, pode-se citar uma rede de interconexão tipo mesh (figura 3.7b).

A organização da memória influi, inicialmente, no tamanho e no tipo, se local ou global. O emprego de memória local diminui a necessidade de comunicação com um meio externo, reduzindo assim o tempo gasto em comunicação e eliminando eventuais gargalos. No caso específico de arquiteturas SIMD a utilização de uma memória local implica em que os dados a serem processados sejam recuperados localmente a cada processador, processados e posteriormente recuperados. Observa-se aqui que a organização dos dados na memória de cada elemento processador deve considerar o tipo de processamento a ser executado e a rede de interconexão utilizada.

Por último, o tipo de operações aritméticas e lógicas a serem realizadas determinam a complexidade do elemento processador. Também por uma questão de implementação física, o elemento processador deve ser o mais simples possível.

Para ilustrar esta dependência entre aplicações e arquiteturas, pode-se analisar certas características apresentadas por classes de aplicações mais apropriadas para máquinas SIMD, como por exemplo, tratamento de imagens, previsão de tempo, processamento sobre grafos, e soluções de sistemas de equações lineares, entre outros.

No exemplo apresentado na seção anterior (3.4), observa-se a necessidade de comunicação de um elemento processador com seus vizinhos nas direções norte, sul, leste, e oeste. Este tipo de comunicação leva ao emprego de uma rede de interconexão mesh entre os processadores. Nota-se, ainda, neste exemplo que a necessidade de comunicação

(granularidade) no caso da avaliação do gradiente de Sobel é pequena, e que no algoritmo de mapeamento de níveis de cinza é nulo. O tipo de operações a serem efetuadas, como soma, subtração, e operações lógicas do tipo AND, XOR, NOT, e OR possibilitam que o elemento processador seja uma simples ULA (Unidade Lógica e Aritmética). Organizando-se os dados na memória local de cada elemento processador, obtem-se um mapeamento 1:1 entre pixels da imagem e elementos processadores. Este mapeamento, aliado ao emprego da rede de inteconexão mesh, torna-se eficaz por já produzir o efeito de "vizinhança-4" necessário à avaliação do gradiente de Sobel. A partir desta breve análise verifica-se que tanto o algoritmo de mapeamento de nível de cinza, como o de gradiente de Sobel, possuem características favoráveis ao emprego em máquinas SIMD.

A seguir são enumeradas outras áreas de aplicação em que as arquiteturas do tipo SIMD apresentam características favoráveis para sua implementação.

Em [GIL 71] é apresentado o emprego de uma arquitetura SIMD visando aplicações em previsão de tempo. Esta aplicação comparada com a anterior necessita de elementos processadores um pouco mais complexos, capazes de efetuarem operações de divisão e multiplicação requeridos no processo. O tipo de comunicação necessário a esta aplicação é uma espécie de superposição de elementos processadores, implicando em uma granularidade elevada. Para realizar de forma eficiente esta comunicação foi desenvolvido uma rede de interconexão específica, denominada de "flip" ("flip network").

O processamento simbólico de informações é uma classe de operações que numa primeira instância parece favorável ao emprego de máquinas SIMD. São exemplos típicos desta classe, algoritmos de "sorting" e manipulação de grafos. Em [CRA 68] é discutido as vantagens do emprego de processadores associativos para a determinação de caminho

mínimo em um grafo, apresentando um algoritmo para exploração das capacidades de pesquisa inerentes às memórias associativas. Orlando e Berra [ORL 72] também abordam o uso de processamento associativo para determinação do problema de custo máximo e mínimo em grafos. Mais recentemente, Paige [PAI 85] propôs uma adaptação, para explorar paralelismo, em algoritmos sequenciais de determinação de caminho mínimo em grafos (Dijkstra e Ford). Ainda sobre operações com grafos cita-se [CAR 85] e [GOP 85]. Esta classe de aplicações caracteriza-se por não necessitar operações aritméticas complexas, sendo sua eficácia determinada pelo algoritmo selecionado, e pela rede de interconexão empregada.

Outra classe de aplicações, onde as máquinas SIMD parecem ser adequadas para explorar um paralelismo inerente, é na solução de sistemas de equações. Notadamente, os fatores que determinam a eficiência das máquinas SIMD nestas operações são: a capacidade aritmética dos elementos processadores, e as facilidades providas pela rede de interconexão para implementação do algoritmo. Em [CAP 85] é apresentado a proposta de uma implementação paralela para avaliação de sistemas lineares, empregando o método de eliminação de Gauss.

# 4 PROPOSTA PARA ARQUITETURA PARALELA MATRICIAL

Neste capítulo é apresentada uma proposta de arquitetura para uma máquina SIMD, tipo processador matricial. As principais diretrizes que orientaram o desenvolvimento da proposta foram: (a) ser viável sua implementação no CPGCC, e (b) utilizar a linguagem GAL (GAPP Algorithm Language) [NCR 85a].

Com o objetivo de permitir a viabilidade na proposta da arquitetura são definidas estruturas que podem ser implementadas com circuitos integrados comercialmente disponíveis no Brasil. Em função desta característica, a simplicidade sempre foi uma preocupação na proposta da arquitetura. O único circuito integrado não encontrado no Brasil é a matriz de processadores (GAPP), cujo pedido de importação já foi providenciado junto ao CNPq. Uma descrição do GAPP é encontrada no anexo 1 deste trabalho.

Um ponto importante, abordado no capítulo 2, é a dificuldade de elaborar de forma eficiente programas para uma máquina paralela. Visando suprir esta lacuna considerou-se na proposta da arquitetura a utilização da linguagem GAL (anexo 2). A principal vantagem no emprego da linguagem GAL reside no fato desta linguagem ter sido definida, pela NCR, especificamente para o GAPP. A linguagem GAL é, ainda, uma linguagem "C-like", o que fornece uma forma eficiente e fácil para programar sistemas baseados em GAPP.

Outra vantagem importante no uso da GAL, é que o compilador da linguagem (CG-compiler GAL) gera código em duas etapas. Na primeira etapa, o programa fonte é analisado verificando-se a sintaxe, como resultado um arquivo intermediário de saída é gerado. Este arquivo intermediário é composto por instruções para um processador virtual, o qual realiza as instruções de fluxo de controle da GAL e por instruções para o GAPP. Na segunda etapa, o arquivo intermediário é lido e as instruções que o compõem são

interpretadas, gerando, assim, as instruções de controle para o sistema. A descrição do formato da linguagem de interpretação, e do arquivo intermediário são fornecidos nos anexos 4 e 5, respectivamente.

No compilador originalmente distribuído pela NCR, o interpretador é voltado para o emprego em um sistema de desenvolvimento, baseado em GAPP, da própria NCR, o GAPSYS. A idéia é explorar todo o ambiente de programação já existente (GAL, compilador, etc...) substituindo o interpretador, segunda etapa do compilador, por um outro que se adapte a arquitetura proposta.

## 4.1 Arquitetura genérica para processadores matriciais

Um processador matricial, de uma forma genérica, pode ser esquematizado na figura 4.1, como sendo composto por 3 partes distintas: máquina hospedeira, unidade de controle e matriz de processadores. A máquina hospedeira, em função da especialização funcional do processador matricial faz-se necessária para o gerenciamento e interfaceamento deste com o meio externo executando as tarefas de mais alto nível do sistema. São operações típicas do hospedeiro: carga do programa a ser executado, determinação de pontos onde o processamento paralelo é requerido, carga e recuperação de dados na matriz de processadores. O hospedeiro é implementado através de uma máquina "general purpose" estando o processador matricial conectado como um subsistema de E/S, comportando-se de forma similar a um coprocessador.

A unidade de controle, por sua vez, provê as instruções a serem executadas pelos processadores que compõem a matriz, determinando o fluxo de controle do programa em função de condições atingidas durante o processamento. Está ainda sob o gerenciamento da unidade de controle o interfaceamento da matriz de processadores com o meio externo.

Por último, a matriz de processadores que é constituída por um número de elementos processadores (PEs), bastante simples, organizados de forma bidimensional, e interligados por algum tipo de rede de interconexão. Geralmente cada PE é composto por uma ULA de 1 bit, e possui interconexões com os processadores vizinhos imediatos. A cada PE, pode ainda, estar associado um conjunto de registradores e uma memória local.

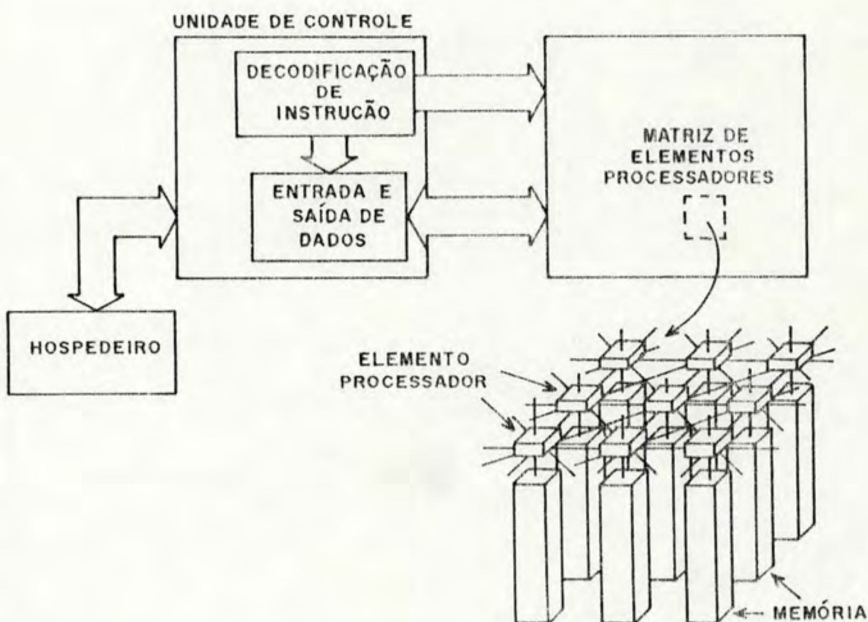



**figura 4.1 - Arquitetura genérica para máquinas matriciais**

## 4.2 Arquitetura do sistema proposto

A arquitetura do sistema proposto segue a organização básica para processadores matriciais apresentada na seção anterior. Sendo assim, o sistema proposto é composto de três partes distintas: máquina hospedeira, matriz de processadores, e unidade de controle. Por uma questão de clareza na apresentação da arquitetura proposta, a unidade de controle é subdividida em quatro partes: interface com o hospedeiro, módulo de controle, módulo de memória, e módulo de reformatação de dados. A figura 4.2 mostra de forma esquemática a arquitetura do sistema proposto.

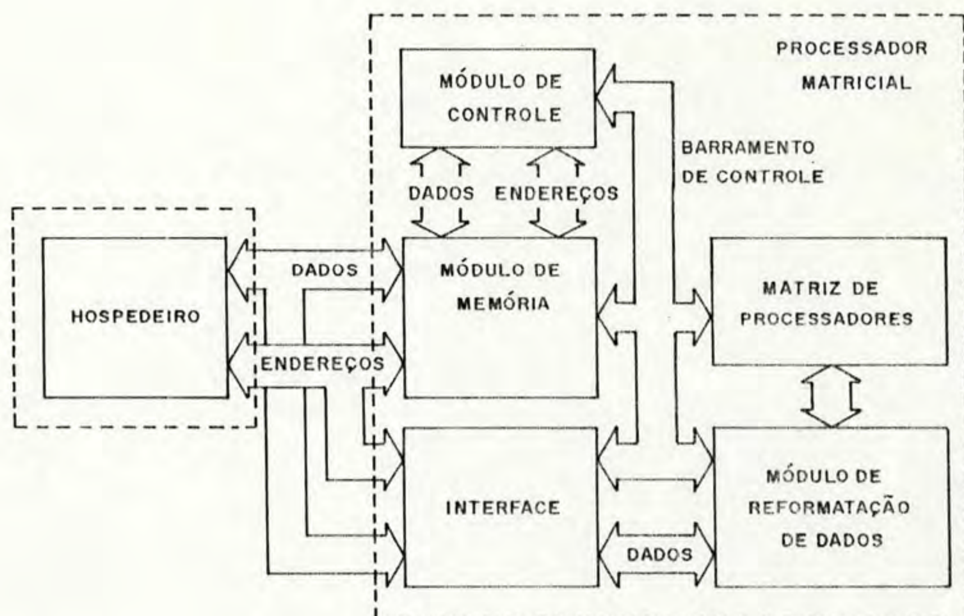



figura 4.2 - Arquitetura do sistema proposto

Por uma questão de disponibilidade, e por ser uma máquina bastante difundida em nossa realidade, a máquina hospedeira será um microcomputador IBM-PC compatível. O processador matricial proposto é então mapeado na memória de E/S do PC, permitindo assim uma fácil interação entre o hospedeiro e o processador matricial.

A interface com o hospedeiro é responsável pela interação entre o processador matricial e o microcomputador IBM-PC. Sua principal função é prover meios eficientes para os procedimentos de carga e recuperação de dados. Este interfaceamento é implementado por um mapeamento das principais unidades funcionais do sistema na memória de E/S do IBM-PC.

A matriz de processadores é composta por 144 elementos processadores organizados em uma matriz de 12x12 (2 chips GAPP). A matriz de processadores é responsável pela execução paralela de programas previamente carregados no módulo de memória. A geração de comandos para a matriz de processadores é responsabilidade do módulo de controle.

O módulo de controle é constituído basicamente por um microprocessador. A função principal do módulo de controle é executar a busca de instruções no módulo de memória, interpretá-las, e executar na matriz de processadores a ação correspondente. É responsabilidade, ainda, do módulo de controle auxiliar nos procedimentos de carga de dados.

O módulo de memória contém exclusivamente os programas a serem executados pela matriz de processadores. Os dados a serem processados são armazenados localmente a cada elemento processador.

O módulo de reformatação de dados faz-se necessário para converter o formato dos dados em função da maneira pela qual o hospedeiro e a matriz de processadores manipulam os dados. Os dados provenientes do hospedeiro são tratados de forma serial por palavra paralelo por bit, enquanto que o GAPP os considera paralelo por palavra serial por bit. Esta tarefa de reformatação é denominada de "corner turning".

A seguir, nas próximas seções, abordar-se-á cada uma destas partes de forma mais detalhada.

### 4.2.1 Máquina hospedeira

A máquina hospedeira escolhida para o sistema proposto é um microcomputador IBM-PC compatível. O processador matricial proposto é interconectado no barramento de dados e endereço do hospedeiro. A interface entre o hospedeiro e o processador é então efetuada através do mapeamento de estruturas do processador matricial em área de memória de E/S do hospedeiro.

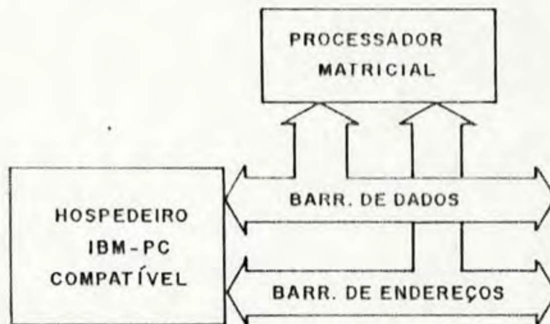



**figura 4.3 - Conexão hospedeiro-processador matricial**

### 4.2.2 Matriz de processadores

A matriz de processadores do sistema proposto, como mencionado anteriormente, é implementada com o auxílio de dois chips GAPP (anexo 1), compondo uma organização de 12x12 elementos processadores.

A arquitetura do circuito integrado GAPP segue uma filosofia RISC, e é microprogramado horizontalmente. Como consequência desta última característica são necessários 20 linhas de controle para a execução de uma instrução: 13 linhas para decodificação da instrução ($C_0$-$c_{12}$), e 7 linhas de endereço ($A_0$-$A_6$) para acesso à RAM local dos elementos processadores.

As instruções (comandos) para a matriz de processadores são fornecidos pela unidade de controle do sistema.

O módulo de controle executa acessos ao módulo de memória para buscar as instruções a serem executadas pela matriz de processadores. Cada instrução é composta por duas partes: comandos ($C_0-C_{12}$) e endereços ($A_0-A_6$). A instrução a ser executada é escrita pelo módulo de controle nos registradores de comando.

Os dados a serem processados são carregados na matriz de processadores através do módulo de reformatação de dados. Este mesmo módulo é responsável pela recuperação dos mesmos após terem sido processados.

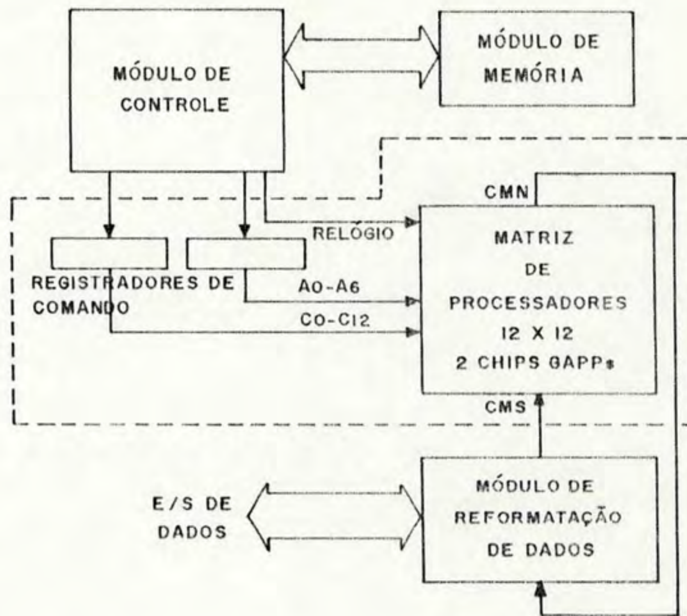



figura 4.4 - Organização da matriz de processadores

### 4.2.3 Interface com o hospedeiro

A interface do sistema proposto com o hospedeiro é executada a partir de um conjunto de endereços da memória de E/S do microcomputador IBM-PC compatível. Esta opção de interfaceamento possilita através do emprego de instruções do tipo IN/OUT (família 8086) uma forma eficiente e prática de interação entre o hospedeiro e a matriz de processadores. Os endereços da memória de E/S que compõem a interface são cinco: três associados a registradores, um associado ao módulo de reformatação de dados, e o último ao módulo de memória.

Os registradores que compoēm a interface são: registrador de comando, registrador endereçador de memória, e registrador de status. No registrador de comandos são escritas palavras de controle que determinam a operação a ser executada pela matriz, tipicamente, inicialização do sistema, carga de programa, carga de dados, e início de operação. O registrador endereçador de memória possui o endereço a partir do qual deseja-se iniciar o procedimento de carga de programa no módulo de memória. E, finalmente, o registrador de status que fornece indicações do estado do sistema, como, executando, pronto para operação (ativo), em "corner turning". A organização de cada um destes registradores é fornecida na figura 4.5.

A associação de um endereço de memória de E/S do IBM-PC compatível ao módulo de reformatação de dados permite que a carga e a recuperação de dados seja efetuada pelo hospedeiro a partir de escritas (OUT), e leituras (IN) neste endereço. Este procedimento permite, ainda, que para algumas aplicações os dados sejam recuperados, ou carregados, paralelamente à execução de programas na matriz de processadores.

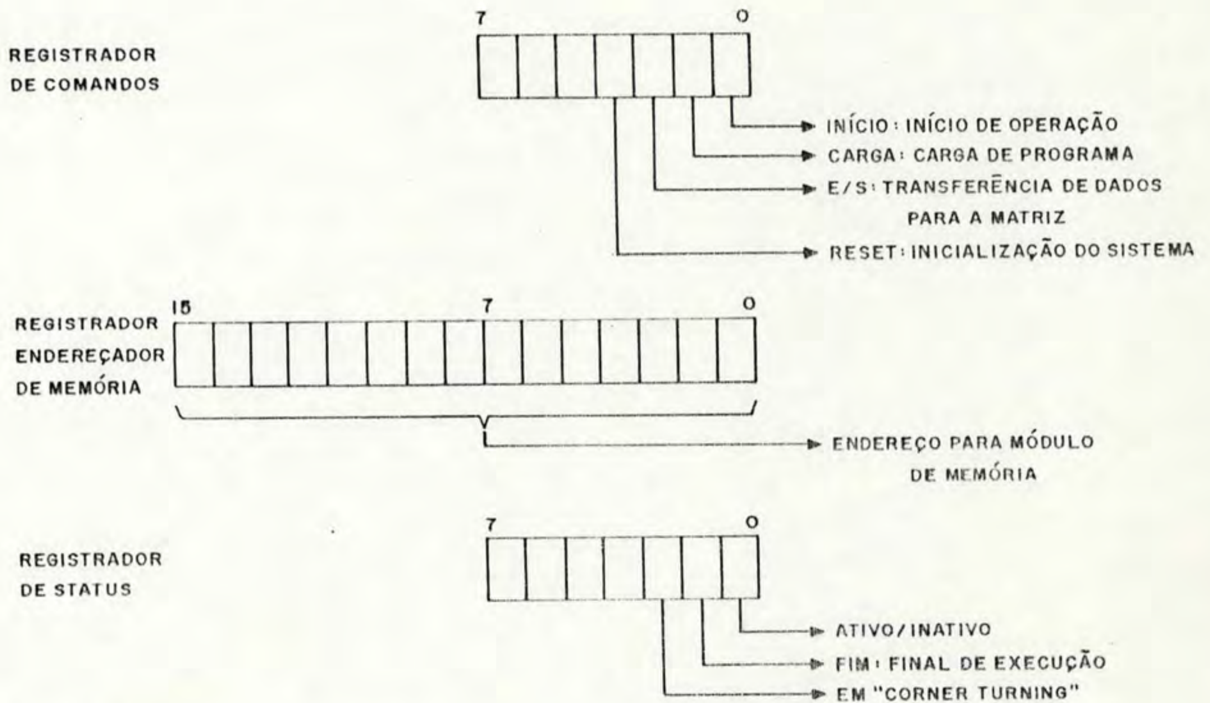



figura 4.5 - Registradores de interface

Para evitar o mapeamento 1:1 entre espaço de memória do IBM-PC compatível, principal ou de E/S, com o módulo de memória, optou-se por acessar este módulo a partir de dois endereços de E/S. O primeiro endereço corresponde ao registrador endereçador de memória, que indica o endereço inicial de carga, permitindo uma versatilidade na escolha do ponto inicial. O segundo endereço, após decodificado, é associado a um sinal de incremento no registrador endereçador de memória, o que possibilita a escrita em posições sucessivas de memória.

UFRGS
BIBLIOTECA
CPD/

### 4.2.4 Módulo de controle

O módulo de controle é responsável pela geração das linhas de controle ($C_0$-$C_{12}$), e endereços ($A_0$-$A_6$) para a matriz de processadores. Este módulo é composto por um microprocessador INTEL 8031 e por uma memória ROM. A escolha do microprocessador 8031 advém de três fatores: (a) baixo custo, (b) versatilidade em operações de E/S através do emprego de portas facilitando operações a bit (testes de condições externas), e (c) possuir uma arquitetura tipo Harward (memória de dados separada da memória de programa). Esta última característica permite que o software de geração de controle seja gravado em ROM, e que o programa a ser executado pela matriz de processadores (dado para o software de controle) seja carregado, via hospedeiro, no módulo de memória.

O software de controle, gravado na ROM, é basicamente um interpretador para o arquivo intermediário gerado pelo compilador GAL (CG). A especificação de quais ações de controle o interpretador deve gerar, e de como são enviados comandos à matriz de processadores são fornecidos no anexo 4.

O módulo de controle possui uma habilitação geral. Esta habilitação é empregada durante o procedimento de carga de programas no módulo de memória pelo hospedeiro para inibir acesso do controle ao módulo de memória. Esta habilitação ocorre através de um bit (bit CARGA) no registrador de comandos. O registrador de comandos possui ainda um bit de INICIO que sinaliza o início de execução de uma tarefa.

O software do interpretador executa acessos ao módulo de memória realizando uma espécie de ciclo de busca de instruções. O interpretador, após buscar a instrução, verifica se é de controle interno do interpretador, ou se é uma instrução para a matriz de processadores. Caso seja de

controle, o software executa a ação apropriada; se for para matriz de processadores, o interpretador a carrega nos registradores de comando e endereços do GAPP para ser executada. Após a carga da instrução o microprocessador, via uma lógica de relógio, habilita a execução de um ciclo pela matriz de processadores.

A matriz de processadores fornece como meio para indicar condições durante a execução de programas a linha de Global Output (GO), a qual é enviada para o 8031. A linha GO pode ser conectada a um bit de uma porta do 8031, ou aos pinos T0 e T1 (contadores de evento do 8031). A linha GO é empregada, através das instruções de controle JMPGOHI e JMPGOLOW do interpretador (anexo 4), para executar desvios condicionais no fluxo de controle do programa em função do estado que esta linha se encontra.

O procedimento de transferência de dados entre o módulo de reformatação de dados e da matriz de processadores é gerenciado pelo módulo de controle. Quando o módulo de reformatação possui dados aptos a serem transferidos para a matriz, o hospedeiro, via registrador de comandos, provoca uma interrupção no 8031. A rotina de atendimento a interrupção efetiva, então, a transferência de dados.

O módulo de controle, ao final da execução de um programa, sinaliza o hospedeiro através de um bit de FIM contido no registrador de status. Fazem parte, ainda, do registrador de status um bit que fornece o estado atual do 8031 (ativo/inativo), e um bit indicativo da operação de "corner turning" em andamento.

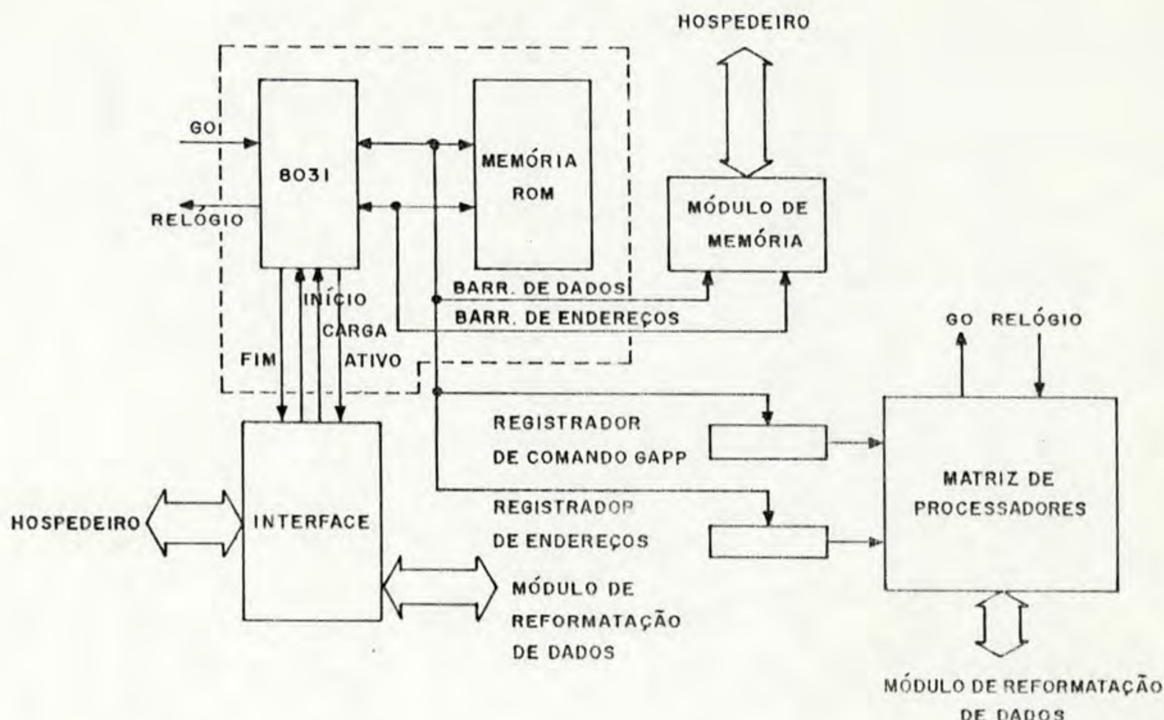



figura 4.6 - Estrutura do módulo de controle

### 4.2.5 Módulo de memória de programa

O módulo de memória é composto por uma memória RAM de 64K, e sua principal função é armazenar o programa a ser executado pela matriz de processadores. A escolha de 64k para a memória está baseado apenas na capacidade máxima de endereçamento do 8031 sem necessitar lógica adicional para acessar maiores porções de memória.

O módulo de memória é acessado por duas fontes distintas: o hospedeiro e o módulo de controle. O hospedeiro acessa o módulo de memória para efetuar a carga do programa a ser executado pela matriz de processadores. Já o módulo de controle executa acessos ao módulo de memória para interpretar o programa, e gerar o controle necessário à matriz de processadores. A seleção de qual módulo irá acessar a memória é realizada por um bit no registrador de comandos (bit CARGA).

Para efetuar a carga de programas no módulo de memória, o hospedeiro, inicialmente, envia uma palavra de

controle ao registrador de comandos colocando em 1 o bit correspondente a carga (CARGA). Este bit quando em 1 desabilita, através de buffers tri-state, o acesso do módulo de controle à memória, conectando-o aos barramentos de endereço e dados do hospedeiro. A partir deste ponto o hospedeiro determina, empregando o registrador endereçador de memória, o ponto a partir do qual o programa será carregado. Após a determinação do ponto inicial de carga, endereço na memória RAM, o hospedeiro executa sucessivas escritas no endereço de memória de E/S correspondente ao módulo de memória. A cada escrita neste endereço (OUT) uma lógica de decodificação causa o incremento do registrador endereçador de memória, permitindo assim a escrita em sucessivas posições de memória. Ao final do procedimento de carga de programa o bit CARGA é resetado, habilitando assim o acesso ao módulo de controle.

A figura 4.7 mostra de forma esquemática a estrutura do módulo de memória.

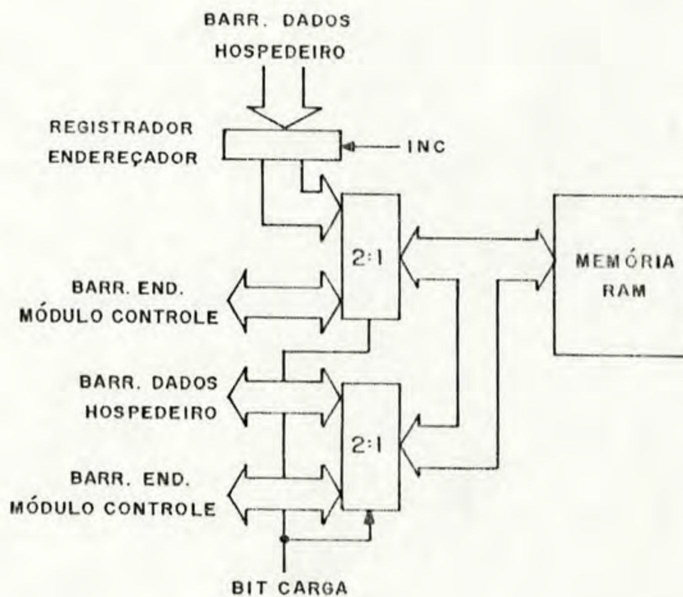



figura 4.7 - Estrutura do módulo de memória

### 4.2.6 Módulo de formatação de dados

Os sistemas digitais, de uma forma geral, empregam como dados um conjunto de valores organizados de forma serial por palavra paralelo por bit, transmitidos as demais unidades que compõem o sistema via barramento de dados. Para operar-se de forma eficiente com o GAPP, os dados na matriz de processadores devem estar associados a cada elemento processador, sendo então armazenados em sua memória local. Para obter-se esta organização de dados em memória, os dados a serem fornecidos, ou mapeados, na matriz de processadores devem ser feitos de forma serial por bit paralelo por palavra. Esta adaptação de formato, onde os dados fisicamente fazem uma rotação de $90^o$ de direção para serem reformatados de paralelo por bit para serial por bit é denominado de operação de "corner turning".

Uma das maneiras para implementar-se a operação de "corner turning" é utilizar um buffer como uma área auxiliar para o armazenamento e transformação do formato de dados. Este buffer é denominado de CTLB, do inglês Corner Turning Line Buffer, e pode ser realizado fisicamente com o auxílio de chips GAPPs, ou com quaisquer dispositivos que possibilitem a transformação do formato de dados, como por exemplo, registrador de deslocamento. A figura 4.8 fornece um esquema para a implementação deste método.

O número de linhas (M), e colunas (N) da matriz de processadores é definido arbitrariamente, em função de requesitos de projeto, mantendo, é claro, a relação de linhas e colunas que um único chip GAPP possui (12x6). O número de colunas no CTLB deve ser o mesmo que o da matriz de processadores, ou seja, "N". A largura "B" dos dados definem o tamanho, isto é, a quantidade de linhas que devem haver no CTLB. A linha de comunicação CMN da matriz de processadores é conectada na entrada sul do CTLB. De forma análoga, a saída norte do CTLB é conectada a entrada sul da matriz de processadores. Este tipo de conexão é denominado

de cilíndrica.

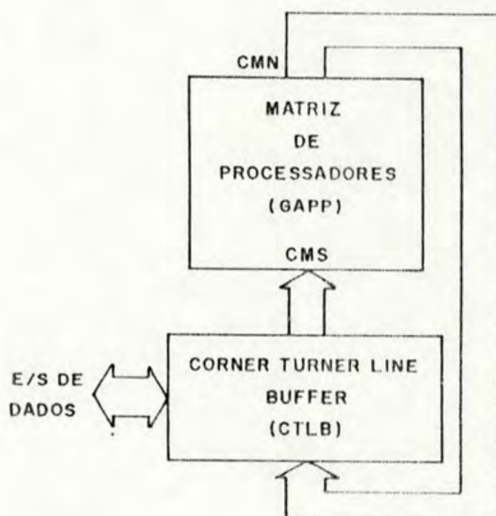



**figura 4.8 - "corner turning" com conexão cilíndrica**

O procedimento de "corner turning", nesta configuração, é descrita de forma algorítmica na figura 4.9. Os dados são inseridos no CTLB um dado a cada instante. Ao ser inserido um dado no instante "t", o dado inserido no instante "t-1" é deslocado à direita, ou à esquerda (depende da ligação empregada), no CTLB. Esta operação ocorre sucessivamente até que "N" dados tenham sido inseridos. Após a inserção de uma linha de dados o conteúdo do CTLB é deslocado para o interior da matriz de processadores via linha de comunicação CMS. Este procedimento é repetido "M" vezes. A retirada dos dados ocorre de forma análoga, só que no sentido inverso.

```
begin
     for i=1 to M do
         begin
         for j=1 to N do
             deslocar dado para interior do CTBL;
         for k=1 to B do
             begin
             (1) carregar bit k-1 dos dados que já estão na
             RAM da matriz para os registradores CM da
             matriz.
             (2) deslocar o conteúdo do CTLB e do
             registrador CM da matriz uma posição para cima
             (norte).
             (3) carregar o bit k-1 dos dados da matriz do
             registrador CM para a RAM.
             end; /*end for k */
         end; /* end for i */
end;
```

**figura 4.9 - Algoritmo de "corner turning" por deslocamento**

Implementando-se o CTLB com chips GAPP, obtem-se a partir da descrição algorítmica acima, o seguinte programa, escrito em GAL (anexo 2), para a realização do "corner turning".

```
corner_turner(dest,X,Y,B)
image dest      ; /* endereco inicial de carga do dado na
                  RAM do elemento processador da matriz*/
int   x,          /* nro de colunas da matriz */
      y,          /* nro de linhas da matriz */
      b;          /* largura dos dados */

{
 int i,j,k;       /* contadores auxiliares */

/*
-----------------------------------------------------------
                Entrada de dados no CTBL
-----------------------------------------------------------
*/

 for(i=0;i<y;i++)        /* operacao (0) */
    {
    for(j=0;j<x,j++) ew:=e ns:=0 cs:=0;
    ram:=sm cm=ram;

/*
-----------------------------------------------------------
Tranfere do CTLB para RAM local dos elementos processadores
-----------------------------------------------------------
*/
 for(k=0;k<b;k++)

    {
    cm:=ram dest:(k); /* conteudo da RAM para CM (1)*/
    cm:=cms;           /* desloca para linha de cima (2)*/
    ram dest:(k):=cm; /* armazena dado na RAM (3)*/
    } /* end k */
    } /* end i */
 }
```

**figura 4.10 - Programa em GAL para "corner turning" por deslocamento**

Observando-se as operações marcadas com (0), (1), (2) e (3) na figura 4.10 pode-se fazer as seguintes afirmações:

(a) a entrada de dados, operação (0), no CTLB independe da matriz de processadores, o que possibilita que a matriz de processadores esteja livre para executar programas enquanto está sendo feita a entrada de dados;

(b) o deslocamento de dados, operação (2), emprega somente o registrador interno de comunicação CM (vide descrição interna do GAPP - anexo 1). O fato do registrador CM não possuir entrada para a ULA também permite que esta operação possa ser executada em paralelo com o programa em execução na matriz de processadores. Esta característica só pode ser explorada se ao escrever-se programas para a matriz de processadores o registrador CM não for empregado em procedimentos de avaliação.

(c) as operações (1) e (3) necessitam da RAM local dos elementos processadores, a qual é empregada na maioria das operações executadas pela matriz de processadores. Como consequência dois ciclos de execução devem ser "roubados" da matriz para a execução destas operações. Observa-se que esta necessidade causa uma degradação na performance de avaliação do programa da matriz em função da largura de bits.

Para a implementação em hardware da operação de "corner turning" existem duas abordagens [NCR 85c]. Na primeira abordagem, são empregados dois controladores, um para a matriz de processadores, e outro para o CTLB. Estes controladores atuam de forma independente sendo que o da matriz controla a execução do programa, e o do CTLB o algoritmo de "corner turning". Nesta abordagem estes controladores, via hardware, devem ser sincronizados para executar em conjunto as operações (1) e (3). Normalmente isto pode ser feito através da detecção destas operações via decodificação das linhas de controle do CTLB. Ao detectar-se as operações (1) e (3) o controlador da matriz é

inibido, e o controle passa a ser executado apenas pelo controlador do CTLB. Esta abordagem é comumente empregada para sistemas que requerem processamento em tempo real.

Na segunda abordagem, emprega-se apenas um controlador, o qual controla a execução do programa na matriz de processadores e a carga de dados no CTLB. Esta abordagem implica que execução e carga ocorram alternadamente no tempo. O chaveamento de execução de programa para "corner turning", e vice-versa, pode ser feita via interrupção. A desvantagem deste método é uma degradação maior na performance, já que a matriz é inibida para operação durante todo o procedimento de "corner turning".

Para o sistema proposto optou-se por uma derivação destas abordagens. A idéia é executar de forma independente a entrada de dados no CTLB (operação 0), e executar a transferência efetiva de dados (operações 1,2 e 3) com o controlador da matriz de processadores. Esta forma de execução causa um degradação menor que no caso de um único controlador, e maior se comparada com a abordagem de dois controladores. Como o controle será executado através de código interpretado, o que já onera a performance, a escolha desta opção não representa uma forte limitação do sistema. O uso de um controlador elimina ainda o emprego, além de outro controlador, de circuitos adicionais necessário à lógica de sincronização.

A estrutura do módulo de reformatação de dados para o sistema proposto é fornecido na figura 4.11. A carga de dados no CTLB é mapeado na memória de E/S, sendo que a decodificação de uma instrução OUT executada pelo hospedeiro atua sobre as linhas de controle do CTLB gerando a instrução ew:=e. Após a carga de dados no CTLB o hospedeiro força uma interrupção na matriz de processadores para executar a transferência efetiva de dados. Esta interrupção sincroniza a operação do CTLB com a matriz. De forma similar, a execução de uma instrução IN pelo hospedeiro também

UFRGS BIBLIOTECA

gera a instrução ew=w, causando a recuperação de dados no CTLB, através da saída do barramento ew (direita do CTLB). Sugere-se, ainda, que as linhas de controle do CTLB sejam mantidas em zero através de "pull-down", o que implica em quando não houver instruções para o CTLB este esteja efetuando a instrução "nop" em todos os seus campos.

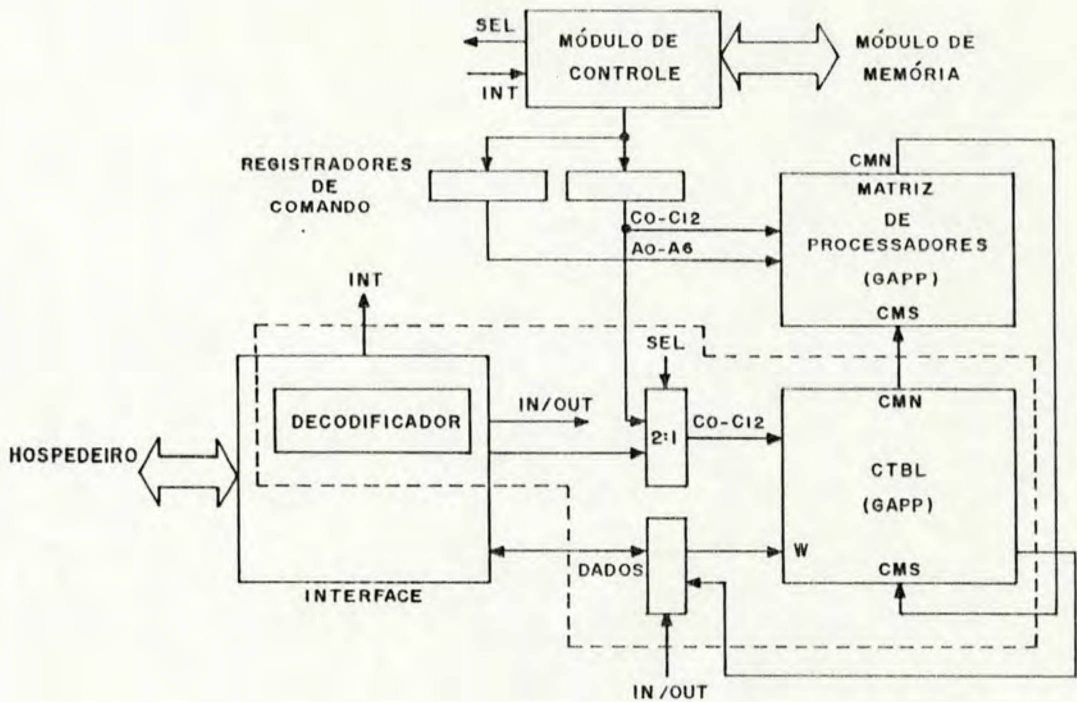



**figura 4.11 - Estrutura do módulo de reformatação de dados**

A interrupção da matriz de processadores é gerada pelo hospedeiro setando-se um bit no registrador de comando (bit E/S). A rotina de atendimento da interrupção chaveia o bit de seleção de linhas de comando do CTLB (SEL), do bloco de decodificação, para serem as mesmas da matriz de processadores, e a seguir executa a rotina de transferência de dados. Ao final da transferência o bit de seleção é resetado.

Em função do método acima descrito, o procedimento de E/S é descrito como segue. O hospedeiro através de instruções OUT envia dados para o CTLB. Após todos dados

serem introduzidos o hospedeiro ativa, via registrador de comando, a transferência de dados executando a rotina fornecida na figura 4.12. Ao final da execução da rotina, os dados que estavam no CTLB estão armazenados na linha inferior da matriz de processadores, e os dados que compunham a linha superior da matriz de processadores estão no CTLB. A recuperação de dados (linha superior) pode ser feita então, pelo hospedeiro, através de sucessivas operações de IN no endereço correspondente ao CTLB.

```
extern image _BUFFER;

corner_t()

{
 int b=size (_BUFFER);        /* contador auxiliar */
 int k;

/*
---------------------------------------------------------------
    Transfere linha do CTL3 para interior da matriz.
---------------------------------------------------------------
*/

 ram:=sm cm=ram;           /* registrador CM recebe conteudo de EW*/
 for(k=0;k<b;k++)
    {
    cm:=ram _BUFFER:(k); /* conteudo da RAM para CM (1)*/
    cm:=cms;             /* desloca para linha de cima (2)*/
    ram _BUFFER:(k):=cm; /* armazena dado na RAM (3)*/
    } /* end k */
  ram=sm ew=ram;         /* registrador EW recebe RAM */
}
```

figura 4.12 - Adaptação da rotina de "corner turning" para o sistema proposto.

## 4.3 Interpretador - o programa de controle

Como apresentado na introdução deste capítulo, a segunda fase do compilador GAL é composta por um interpretador que recebe como entrada um arquivo intermediário gerado pela primeira fase. Este arquivo intermediário é composto por um conjunto de endereços e instruções para o chip GAPP, e por instruções para um processador virtual orientado a pilha.

O processador virtual é na verdade o interpretador que compõe a segunda fase do compilador. O funcionamento do interpretador é bastante simples: ao receber o arquivo intermediário interpreta as instruções gerando o controle ao fluxo do programa, fornecendo como saída os endereços e instruções para o GAPP.

A organização do processador virtual é baseado em três estruturas em memória, (código, variáveis, e pilha), e por cinco registradores internos, empregados no gerenciamento destas estruturas. As próximas seções fornecem uma descrição mais detalhada de cada um destes componentes, e de sua interação.

### 4.3.1 Estruturas de memória

As três estruturas em memória são organizadas a palavra. O conteúdo da memória de código é composto pela parte do arquivo intermediário que possui as instruções do GAPP e do processador virtual.

A memória de variáveis é responsável pelo armazenamento dos valores das variáveis do tipo int que fazem parte do programa GAL. A cada uma das variáveis do programa está associada um endereço nesta estrutura.

Quanto a estrutura de memória de variáveis convém salientar algumas particularidades. Um mesmo endereço na memória de variáveis pode corresponder a mais de uma va-

riável, devido ao fato que as variáveis podem ser locais existindo apenas no trecho de programa que são declaradas. As variáveis do tipo **global** e **extern** são atribuídas a endereços absolutos da memória de variáveis, enquanto que as variáveis do tipo **auto** (local) são atribuídas a endereços relativos ao conteúdo do registrador interno **autopointer**.

Por último, a memória de pilha que é empregada na execução das operações aritméticas e lógicas do processador virtual. As operações são realizadas sobre os conteúdos do topo da memória de pilha, existindo instruções específicas para transferir valores das variáveis da memória para a pilha e vice-versa.

### 4.3.2 Registradores Internos

O gerenciamento das três estruturas de memória é efetuado a partir de cinco registradores: **program counter**, **stack pointer**, **argument pointer**, **auto pointer**, e **address pointer**. O registrador **program counter** aponta para o endereço da memória de código que contém a próxima instrução do arquivo intermediário a ser interpretada. O registrador **stack pointer** mantém como conteúdo o endereço da memória de pilha correspondente ao topo da pilha. O registrador **argument pointer** possui o endereço da memória de pilha onde inicia o armazenamento dos argumentos passados para subrotinas. O registrador **auto pointer** mantém o endereço da memória de variáveis a partir do qual podem ser armazenados variáveis do tipo **auto int** (vide anexo 2). Por último, o registrador **address pointer** que contém o endereço a partir do qual a memória local, de cada elemento processador que compoêm do GAPP, está disponível. Esta informação é útil quando do uso de variáveis do tipo **auto image** (vide anexo 2) em subrotinas. O funcionamento de cada um destes registradores é apresentado a seguir.

O registrador **program counter** é empregado para a busca de instruções do processo de interpretação. O procedimento de interpretação inicia com a busca de uma instrução no endereço da estrutura de memória de código fornecido no **program counter**. A cada ciclo de busca o **program counter** é incrementado para apontar o endereço da próxima instrução na memória de código.

O registrador **stack pointer**, por sua vez, é empregado para acessar o conteúdo do topo da memória de pilha. Como no processador virtual as operações aritméticas e lógicas são executadas sobre operandos que estão no topo da pilha, o registrador **stack pointer** exerce importância fundamental para a execução destas operações.

Os demais registradores são empregados na implementação de chamada a subrotinas, e na utilização de variáveis do tipo **auto**. Para entender o funcionamento destes registradores analisar-se-á o que ocorre na chamada de uma subrotina.

Quando uma subrotina é chamada, os argumentos são inseridos no topo da pilha seguido por uma palavra que fornece o número de argumentos utilizados. Como pode haver chamadas de subrotinas dentro de subrotinas, o valor do registrador **argument pointer** da rotina que executa o CALL é empilhado para que não se perca a informação de onde seus próprios argumentos são armazenados.

O procedimento de chamada de subrotinas determina quantos endereços na memória de dados serão ocupados pelas variáveis **auto int**. De forma similar, é conhecido quantos endereços na memória do GAPP são necessários para as variáveis **auto image**. Em função do número de endereços ocupados por estas variáveis, os registradores **auto pointer** e **address pointer** são, respectivamente, incrementados.

O procedimento executado a seguir é empilhar duas vezes o valor constante (zero) no topo da pilha. Este

procedimento possui a finalidade de reservar espaço para a subrotina retornar um valor inteiro. A seguir o valor do registrador **program counter** é empilhado, sendo então substituído pelo endereço da primeira instrução da subrotina.

A primeira tarefa que uma subrotina executa quando ela é chamada é copiar o conteúdo do **stack pointer** para o **argument pointer**. Desta forma, é salva a posição onde estão armazenados os argumentos da rotina que executou a instrução CALL. Quando, ao final de uma subrotina é executada uma instrução RETURN, o valor de retorno é copiado na posição zerada na pilha no início do procedimento, e o valor do **argument pointer** é copiado para o **stack pointer**. Ao executar este procedimento a pilha se encontra no mesmo estado que estava quando a subrotina foi chamada. O antigo conteúdo do **program counter** é desempilhado e escrito no registrador **program counter**. Este procedimento faz com que a próxima instrução a ser executada seja a seguinte após a chamada da subrotina.

Ainda, ao terminar a execução de uma subrotina os registradores **auto pointer** e **address pointer** são decrementados pelos mesmos valores nos quais eles foram incrementados antes da chamada da subrotina. O valor inteiro (int) de retorno da subrotina é desempilhado e salvo, sendo atribuído, se for o caso, ao nome da subrotina. O valor antigo do **argument pointer** antes da chamada da subrotina e desempilhado e escrito no registrador **argument pointer**, a seguir o número de argumentos é desempilhado, e o **stack pointer** é decrementado por este valor. Neste momento a pilha se encontra no mesmo estado que estava antes da chamada da subrotina.

### 4.3.3 Implementação do Interpretador

Conforme visto na seçao 4.2.4, o módulo de controle é composto por um microprocessador 8031 [INT 81], e uma memória ROM (64K). O módulo de controle executa acessos ao módulo de memória constituído basicamente por uma memória RAM. Na memória RAM, após o procedimento de carga, via carregador, são mantidos o programa a ser interpretado, as estruturas de memória e pilha do interpretador. A memória ROM possui o código que implementa o interpretador.

O programa do interpretador implementa o comportamento descrito anteriormente na seção 4.3.2. As ações de controle tomadas na interpretação de cada uma das instruções que compoêm o arquivo intermediário são descritas no anexo 5.

Com o intuito de validar o funcionamento do processador matricial proposto, foi desenvolvida uma versão para o interpretador (vide anexo 6) empregando-se a linguagem C no ambiente Turbo C [BOR 87].

O interpretador é basicamente composto por um comando **switch** dentro de um laço do tipo **while**. A função do **while** é executar acessos à memória de código, realizando buscas de instruções, até ser atingido o final do programa, sinalizado pela existência de uma instrução **PROGEND**. O dado lido na memória de código (instrução para o processador virtual) é utilizado como expressão de teste do comando **switch**, estando associado a cada instrução um **case**, o qual implementa as ações realizadas por estas sobre as três estruturas de memória e nos registradores internos.

As três estruturas de memória são implementadas com o auxílio de vetores unidimensionais de inteiros.

Associado ao interpretador existe uma rotina de atendimento a interrupção, necessária a transferência de dados entre o módulo de reformatação de dados e a matriz de

processadores. Esta rotina é executada sempre que o bit de E/S do registrador de comandos for setado. A rotina de atendimento a interrupção, basicamente simula a ação executada por uma instrução **CALL** do processador virtual. A rotina chamada pela **CALL** simulada executa o procedimento de "corner turning" dado na figura 4.12.

O interpretador, durante sua fase de inicialização, necessita saber alguns valores que são determinados apenas após o procedimento de compilação e ligação do programa a ser interpretado. Estes valores são: o endereço inicial da rotina de "corner turning", o endereço inicial na RAM local de cada elemento processador onde os dados serão armazenados, o tamanho (largura) da variável image deste dados, a quantidade de endereços utilizados por variáveis **global** e **extern** do tipo **int** e **image**, e o endereço inicial do programa (main). Quem determina estes valores é o programa carregador que será descrito na seção 5.3. A passagem destes valores do carregador para o interpretador se faz através de uma estrutura de comum acesso a estes dois programas: o módulo de memória do processador matricial. O carregador durante o procedimento de carga de programa no processador matricial escreve estes valores nas seis primeiras posições da área destinada à estrutura de pilha. O interpretador ao inicializar sua execução recupera estes valores e os atribui aos respectivos registradores internos.

O endereço inicial da rotina de "corner turning" mais os valores de endereço inicial na RAM local e a largura do dado são necessários para executar-se a CALL simulada. Estes valores durante a inicialização devem ser copiados da área de pilha para variáveis auxiliares, liberando assim as posições ocupadas na pilha.

Na próxima posição da pilha está armazenado o número "n" de variáveis do tipo **global** e **extern** empregados no programa. Estas variáveis ocupam os endereços de 0 a "n-

1" da estrutura de memória de variáveis. O registrador **auto pointer** é então carregado com o próximo endereço da memória de variáveis, ou seja, "n". Os endereços de "n" até o fim da memória de dados são disponíveis para variáveis do tipo **auto int**. De forma similar, o registrador **address pointer** recebe o valor a partir do qual a memória local de cada processador do GAPP está disponível para o emprego de variáveis do tipo **auto image**. Este endereço é fornecido na posição seguinte de memória da estrutura de pilha. A última posição contém o endereço do início do programa, isto é, o endereço da rotina main, o qual é atribuído ao registrador **program counter**. Finalmente, os registradores **argument pointer** e **stack pointer** são zerados.

Quando da implementação física do processador matricial, embora o interpretador tenha sido escrito usando-se C padrão Kernighan e Ritchie, e exista um compilador C para o 8031, sugere-se que seja implementada uma nova versão para o interpretador em assembly 8031. Esta necessidade advém de dois fatores: (a) o código gerado pelo compilador C do 8031, de forma "ad hoc", sabe-se ser bastante ineficiente, e (b) os modelos de organização de memória de dados deste compilador não são muito versáteis, desperdiçando área de memória.

Convém salientar certos cuidados a serem tomados na versão assembly 8031 do interpretador. Primeiro, as estruturas em memória são organizadas a palavras, enquanto que o 8031 executa acessos somente a byte. Segundo, no modelo do interpretador as três estruturas de memória possuem endereço inicial zero, e na versão assembly, em função do mapa de memória a ser empregado para cada área, o endereço inicial é diferente de zero. Deve-se então considerar, no procedimento de inicialização dos registradores internos o valor inicial destas estruturas na memória RAM do processador matricial. Os registradores **argument pointer** e **stack pointer** são inicializados com o endereço inicial da área

destinada a estrutura de pilha. O acesso a estrutura de variáveis é realizada somando-se ao endereço de cada variável o endereço inicial da área destinada a estrutura de variáveis.

U F R G S
BIBLIOTECA
CPD

# 5 AMBIENTE PARA DESENVOLVIMENTO DE PROGRAMAS

O ambiente de programação mínimo necessário para o desenvolvimento de programas no sistema proposto é composto por: um compilador C, um editor de texto, o compilador GAL, programa carregador, e um programa fixo de E/S.

O compilador C é necessário para o desenvolvimento dos programas a serem executados pela máquina hospedeira. A classe de programas que são executados na máquina hospedeira é composta por programas de gerenciamento e controle do processador matricial, realizando, tipicamente, tarefas relacionadas com carga de programas, carga/recuperação de dados, envio e monitoração de palavras de controle ao processador matricial. O editor de textos, por sua vez, é utilizado na criação de arquivos fontes em GAL. O compilador GAL é necessário para a compilação e ligação de programas fontes escritos em GAL. O programa carregador é responsável pela carga do programa a ser executado pela matriz de processadores, e pode, eventualmente, fazer parte de um procedimento de inicialização do programa executado pelo hospedeiro. Por último, tem-se o programa de E/S, que é uma rotina fixa escrita em GAL, responsável pela operação de "corner turning" no sistema proposto. O programa de E/S deve ser sempre ligado ao programa fonte a ser executado na matriz de processadores.

A princípio, o emprego de um compilador C, não é uma necessidade básica do sistema, podendo ser utilizada no desenvolvimento de programas para o hospedeiro qualquer outra linguagem. O fato de recomendar-se o emprego de compilador C, como por exemplo o Turbo C [BOR 87], advém dos seguintes fatores: (a) este compilador inclui um editor para escrita de programas, (b) facilita a inclusão e a chamada do programa carregador no programa a ser executado pela máquina hospedeira, já que ambos são desenvolvidos em C, e (c) facilidade de acesso a endereços de E/S, e manipulação de bits o que permite o envio de palavras de controle

ao processador de forma prática.

A seguir são apresentados o funcionamento do compilador GAL, do programa de E/S, e do programa carregador. Ao final deste capítulo são fornecidos os principais passos a serem executados para o desenvolvimento de programas no sistema proposto.

## 5.1 Compilador GAL - CG

O compilador GAL (CG) recebe como entrada um arquivo fonte escrito na linguagem GAL e fornece como saída, em função de opções de compilação, quatro arquivos distintos: um arquivo intermediário (.SIF), um arquivo intermediário ligado (.LNK), dois arquivos de saída (.OBJ e .G) em formato binário ou ASCII. O compilador, é ainda, organizado em dois módulos, conforme pode ser visto na figura 5.1.

Na primeira fase, o arquivo fonte é compilado verificando-se a sintaxe dos comando gerando o arquivo (.SIF). A segunda fase do compilador é composta pelo ligador e pelo interpretador. Como um programa em GAL pode ser composto por vários módulos compilados independentemente, o ligador faz-se necessário para a partir destes vários módulos, arquivos (.SIF), obter um único módulo executável, arquivo (.LNK).

O arquivo executável (.LNK), ou (.SIF), caso haja um único módulo, é composto por um conjunto de instruções que são dados para o interpretador. O interpretador lê estes arquivos, executa ações de controle relativa a instrução lida e produz um arquivo de saída (.OBJ), ou (.G), em função do formato especificado pela opção de compilação, que contém endereços e comandos para o GAPP.

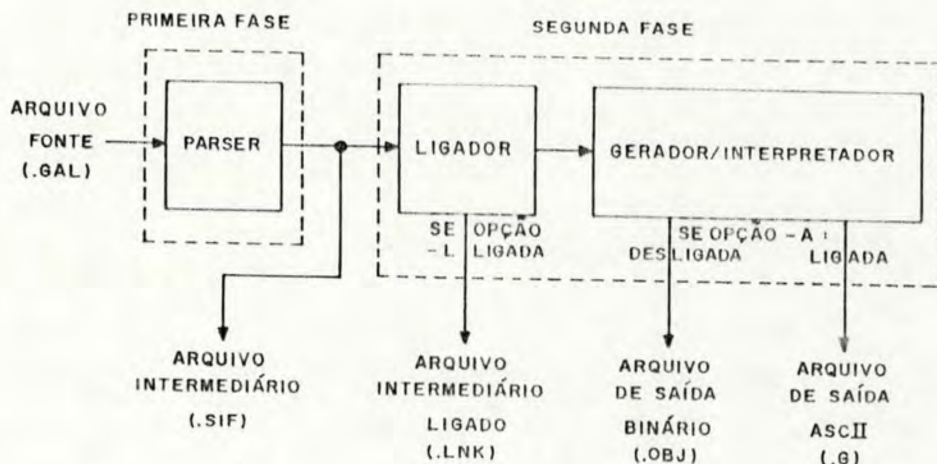



figura 5.1 - Organização do compilador GAL

O compilador GAL distribuído pela NCR foi projetado para operar em conjunto com o sistema de desenvolvimento GAPSYS, também da NCR. Como consequência, o interpretador é dependente da estrutura de hardware que compõe o GAPSYS. Para empregar-se o compilador GAL da NCR no sistema proposto é necessário, via opção de compilação, não executar a parte correspondente ao interpretador/gerador. A interpretação neste caso será feita pelo interpretador do módulo de controle do sistema proposto (seção 4.3).

O arquivo fonte, em GAL, é obtido a partir de qualquer editor de texto em modo não documento. A chamada do compilador é executada através do "shell" do ambiente DOS, digitando-se uma linha de comando correspondente a operação desejada de compilação. O anexo 3 fornece a forma geral da linha de comando, e apresenta, também, as opções de compilação existentes.

## 5.2 CORNER.SIF - o programa E/S

Conforme apresentado na seção 4.2.6, os dados antes de serem operados no processador matricial devem sofrer um procedimento de reformatação. Quem executa esta tarefa é o módulo de reformatação, que sob controle da máquina hospedeira, após receber um conjunto de dados, os transfere para os elementos processadores que compõem a matriz. Esta transferência é realizada executando-se uma rotina de E/S.

A rotina de E/S deve, então, respeitar as considerações sobre como a reformatação de dados (procedimento de "corner turning") deve ser executado no processador matricial proposto, em função de suas características estruturais (arquitetura). Pode-se concluir, como consequência imediata desta vinculação, que o procedimento de "corner turning" é sempre o mesmo, independendo da aplicação. Conclui-se, ainda, que a rotina de E/S deve estar sempre presente em qualquer aplicação escrita para o sistema. Para evitar que seja sempre responsabilidade do usuário a escrita de uma rotina para "corner turning", optou-se pela inclusão de uma rotina, a CORNER.SIF, no ambiente de programação proposto. Esta rotina foi desenvolvida junto com a proposta da arquitetura, e é disponível ao usuário, como se fizesse parte de uma biblioteca de funções do sistema.

O programa de E/S é então uma rotina em GAL, previamente compilada, que executa o procedimento de "corner turning" para o sistema proposto. Salienta-se, ainda, que sua implementação é a mesma fornecida na figura 4.12. Sob o ponto de vista usuário, esta informação é importante, porque diz respeito de como os dados devem ser fornecidos ao sistema para um correto funcionamento: uma linha de dados a cada instante.

Para empregar esta facilidade, o usuário deve incluir no desenvolvimento de seu programa a declaração de uma variável global do tipo image denominada de _BUFFER. Esta variável determina o ponto inicial e final da RAM interna de GAPP onde a CORNER.SIF armazenará os dados introduzidos na matriz. É de responsabilidade do usuário a definição do ponto inicial e do tamanho dos dados (ponto final). Uma forma geral para esta declaração é:

```
image _BUFFER:end_i:end_f;
```

onde,

end_i: endereço inicial da RAM interna do GAPP.
end_f: endereço final da RAM interna do GAPP.

Deve-se, ainda, antes de carregar o programa no processador matricial, arquivo .LNK, ligar o arquivo .SIF correspondente ao programa desenvolvido com o arquivo CORNER.SIF. Este procedimento é efetuado através da seguinte linha de comando do compilador.

```
CG -L nome_arq.SIF CORNER.SIF -O nomer_arq.LNK
```

## 5.3 Carregador

O programa carregador é responsável pela leitura e posterior carga do arquivo intermediário (.LNK) no módulo de memória do processador matricial. O arquivo intermediário é formado por cinco partes distintas: header do arquivo, tabela de símbolos, tabela de inicialização, tabela de rótulos (label), e tabela de código. O header possui informações sobre a estrutura do resto do arquivo, e mantém informações para o processo de ligação. A tabela de símbolos mantém dados a respeito dos nomes de variáveis e subrotinas empregadas no programa. A tabela de inicialização contém os valores atribuídos a variáveis antes da execução do programa (constantes), e são armazenados em endereços associados a variável inicializada na estrutura de memória de dados do interpretador (seção 4.3). A tabela de rótulos fornece a associação de endereços aos rótulos (labels) utilizados no programa. Finalmente, tem-se a tabela de códigos que é composta pelo programa, já compilado, a ser executado pelo processador virtual. O anexo 5 descreve de forma detalhada os componentes de cada uma destas partes.

O carregador inicia sua operação executando uma leitura no header do arquivo. A partir das informaçẽs lidas no header determina-se o tamanho das demais estruturas que compõem o arquivo. No header estão, ainda, alguns valores importantes a serem consideradas na inicialização do interpretador, como o endereço inicial do programa (main), e a quantidade de posições empregadas na memória de dados e RAM local de cada elemento processador.

A partir da tabela de símbolos do arquivo .LNK, obtem-se dados referentes a rotina de "corner turning", como o endereço inicial e final da área destinada ao armazenamento de dados. O espaço ocupado na RAM pelo procedimento de "corner turning" é fornecido a partir de dois valores: o endereço de início e a largura do dado. Estes

valores são necessários para o interpretador executar a simulação de uma instrução CALL quando interrompido para efetuar o "corner turning".

A passagem destes valores, do carregador para o interpretador, é realizada através da área destinada à estrutura de pilha do interpretador. O carregador ao determiná-los, escreve na ordem dada a seguir, nas posições de 0 a 5 da estrutura de pilha, os valores: endereço inicial da rotina de "corner turning", endereço inicial da RAM local, tamanho (largura) do dado, quantidade de posições ocupadas por variáveis globais e externas do tipo **image**, quantidade de posições ocupadas por variáveis globais e externas do tipo **int**, e o endereço da rotina **main**. O interpretador durante sua inicialização, conforme descrito, na seção 4.3.3, recupera estas informações atribuindo-as às respectivas estruturas de controle.

A tabela de inicialização é lida logo a seguir, carregando na estrutura destinada a memória de variáveis os valores correspondentes as variáveis inicializadas.

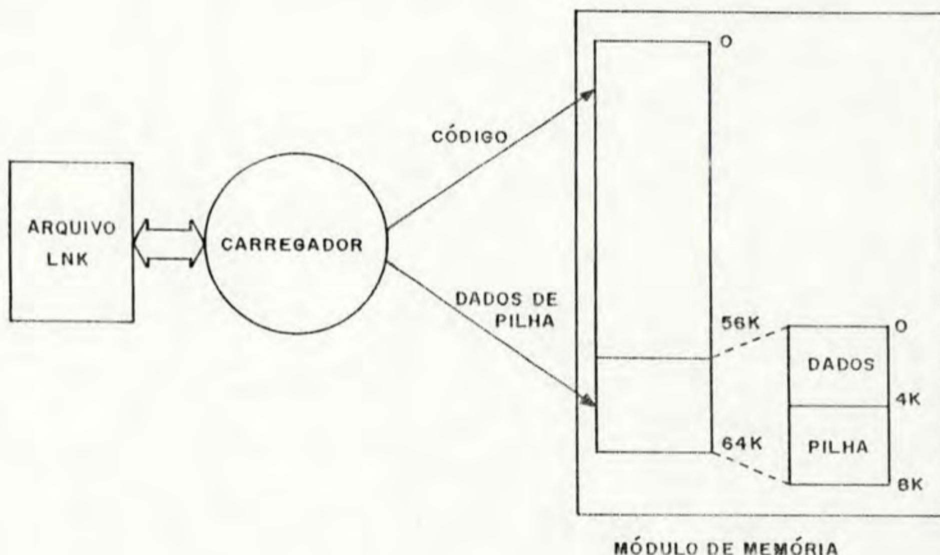



figura 5.2 - Mapeamento de memória

UFRGS BIBLIO

Finalmente, é realizada a leitura na tabela de códigos, transferindo o programa área para a área de memória a esta correspondente (estrutura de código). O mapeamento da área de memória de dados do processador matricial com as estruturas de código, variável e pilha, sugeridos para o interpretador é fornecido na figura 5.2. Salienta-se que este mapeamento deve ser respeitado na implementação do interpretador (versão assembly 8031) do processador matricial.

## 5.4 Desenvolvimento de programas no sistema proposto

O sistema proposto, sob ponto de vista do usuário, é composto por duas partes: máquina hospedeira e processador matricial. Esta característica leva a necessidade, ao desenvolver-se programas, que estes também sejam escritos em duas partes, uma para a máquina hospedeira, outra para o processador matricial. O programa a ser executado pelo processador matricial é um algoritmo paralelo (aplicação). Por sua vez, o programa a ser executado pela máquina hospedeira é responsável por funções de mais alto nível, gerenciando a interface entre o processador matricial com o meio externo (máquina hospedeira).

Os passos necessários ao desenvolvimento de programas para o processador matricial são descritos a seguir. Inicialmente, deve ser criado em um editor de texto, em modo não-documento, um arquivo fonte escrito em GAL que implementa o algoritmo desejado. Após editado deve-se gerar o arquivo intermediário (.SIF) correspondente. Isto é realizado compilando-se o arquivo fonte com a opção -S do compilador GAL. O arquivo (.SIF) gerado neste primeiro passo deve ser então ligado com o arquivo CORNER.SIF, permitindo, assim, a operação de E/S no processador matricial ("corner turning"). Esta ligação também é efetuada com o auxílio do compilador GAL, empregando-se a opção -L. Como resultado do procedimento de ligação obtem-se um arquivo

(.LNK) o qual deve ser carregado no módulo de memória do processador matricial.

A estrutura do programa a ser executado pela máquina hospedeira é mostrada de forma esquemática na figura 5.3. Inicialmente, o programa carregador (seção 5.3) é chamado para efetivar a carga no módulo de memória do processador matricial do programa a ser executado (arquivo .LNK). Após a realização da carga do arquivo .LNK inicia a execução do programa de controle, responsável pelo gerenciamento das operações de mais alto nível entre a máquina hospedeira e o processador matricial. São exemplos de funções típicas do programa de controle, a determinação das operações a serem executadas pelo processador matricial, e a carga e recuperação de dados.

**figura 5.3 - Estrutura geral de um programa para o sistema proposto**

Considerando-se, como exemplo deste procedimento, que se tenha uma imagem de dimensão TAM_X por TAM_Y, sobre a qual executar-se-á algum tipo de processamento. O programa a ser executado pela máquina hospedeira é apresentado na figura 5.4. Observa-se que este programa é desenvolvido em C, o que facilita a sua compilação com o código já compilador do carregador.

```
#include <carrega.h>
   .
   .
#define NRO_LIN 12          /*numero de linhas na matriz de processadores*/
#define NRO_COL 12          /*numero de colunas na matriz de processadores*/
#define TAM_X 64            /*tamanho da imagem a ser processada direcao "x"*/
#define TAM_Y 64            /*tamanho da imagem a ser processada direcao "y"*/
   .

        /* ------   Descricao dos Registradores de Interface -----

        REG_COM  bit_0: inicializa a execucao do interpretador
                 bit_1: seleciona carga - modulo de memoria acessado pelo hospedeiro
                 bit_2: ativa procedimento de "corner turning"
                 bit_3: inicializacao do sistema
                 bit_4 - bit_7: nao usados

        REG_STAT bit_0: modulo de controle em operacao
                 bit_1: fim de execucao do programa
                 bit_2: modulo de controle executando "corner turning"
                 bit_3 - bit_7: nao usados

        REG_END  bit_0 - bit_15: endereco inicial de carga do modulo de memoria

        */

     void main(void)
     {
01   int iter_lin, iter_col, i, j, k, l, k_aux, l_aux;
02   int imagem[TAM_X][TAM_Y];
03   div_t x;

04   le_imagem(imagem);                         /*carrega a imagem*/
     /* Em funcao do tamanho da imagem e da matriz de processadores
        determinar o numero de sub-imagens */
05   x=div(TAM_Y,NRO_COL);
06   iter_col=x.quot+((x.rem)0)?1:0);
07   x=div(TAM_X,NRO_LIN);
08   iter_lin=x.quot+((x.rem)0)?1:0);

09   outportb(REG_COM,0x02);                    /*seleciona acesso ao módulo de memoria*/
10   carrega();                                 /*carrega programa a ser executado*/
11   outportb(REG_COM,0x08);                    /*desabilita carga e inicializa o sistema*/
12   while(!(inportb(REG_STAT)&0x01));          /*espera inicializacao ser completada*/

/*
-------------------------------------------------------------------------------
                    Inicio do programa de controle
-------------------------------------------------------------------------------
*/
13   for(i=0;i<iter_lin;i++)
14      for(j=0;j<iter_col,j++)
15         {   /* carrega uma janela da imagem para processamento */
16         k_aux=i*NRO_LIN;
17         for(k=k_aux;k<k_aux+NRO_LIN;k++)
18            {
19            l_aux=j*NRO_COL;
20            for(l=l_aux;l<l_aux+NRO_COL;l++)
21               if ((k<TAM_X) & (l<TAM_Y)) outportb(CTLB,imagem[k][l]);   /*carrega linha da sub-imagem no CTLB*/
22            outportb(REG_COM,(inportb(REG_COM)||0x02);                 /*dispara procedimento de "corner turnin
23            while (!(inport(REG_STA) & 0x04);                          /*espera fim do "corner turning"*/
24            outportb(REG_COM,(inportb(REG_COM)||0x0FD);                /*reseta bit de operacao de "corner turn
25            for(l=l_aux;l<l_aux+NRO_COL;l++)
26               if((K<TAM_X) & (l<TAM_Y)) imagem[k,l]=inportb(CTLB);       /*recupera dados do CTLb*/
27            }
28         outportb(REG_COM,(inportb(REG_COM)||0x01);                    /*inicializa a execucao do programa*/
29         while(!(inportb(REG_STST) & 0x02);                            /*espera o fim da execucao*/
30         }
31   }
```

figura 5.4 - Exemplo de utilização do processador matricial

Inicialmente, a máquina hospedeira executa um acesso a um meio físico onde a imagem está armazenada (linha 4). Nas linhas 5 e 8, é calculado, em função da dimensão da imagem em relação a dimensão da matriz de processadores, em quantas sub-imagens a imagem original precisa ser dividida para ser capaz de ser processada na matriz de processadores. A seguir, linha 9, via registrador de comandos, o acesso ao módulo de memória é dado para o barramento da máquina hospedeira possibilitando a carga do programa. Na linha 10 é executada a chamada ao carregador, o qual solicita o nome do programa (arquivo .LNK) a ser executado e efetua sua carga no módulo de memória. É responsabilidade do carregador, conforme apresentado na seção 5.3, a definição de valores necessários a inicialização do sistema. Na linha 11, o módulo de memória passa a ser acessado pelo módulo de controle do processador matricial, e sistema é inicializado. Por último, na linha 12, espera-se que a inicialização tenha sido completada.

Após a inicialização do sistema, entra em execução a parte do programa responsável pela divisão, carga e recuperação das sub-imagens no processador matricial. O primeiro laço **for** (linha 13-31) determina o número de sub-imagens necessárias na dimensão Y (número de linhas) da imagem. O segundo laço **for** (linha 14-30), por sua vez, determina o número de subdivisoẽs necessárias na dimensão X (número de colunas). Para cada iteração dos laços **for** das linhas 13 e 14, uma sub-imagem é carregada na matriz de processadores, via "corner turning", processada e depois recuperada. Este procedimento é realizado enquanto houver subimagens.

O procedimento executado para cada sub-imagem é descrito a seguir. Inicialmente, a sub-imagem deve ser carregada na matriz de processadores via "corner turning". A carga da sub-imagem é realizada nas linhas 17 a 27, a partir da execução dos seguintes passos: (a) introdução de

uma linha no CTLB (linhas 20-21); (b) disparo da função de "corner turning" (linha 22); (c) espera do final da transferência de dados do CTLB para a matriz (linha 23); (d) retirada do procedimento de "corner turning" (linha 24); e (e) recuperação da linha do CTLB (linhas 25-26). Observa-se que devido a maneira pela qual o "corner turning" é implementado, ao introduzir-se dados na linha inferior da matriz de processadores, a linha superior da matriz é recuperada. Após a carga da sub-imagem é disparado o início da execução do programa (linha 28). Por último, na linha 29, é realizado um ciclo de espera de fim de processamento. Estes passos são repetidos até que toda a imagem tenha sido processada.

# 6 VALIDAÇÃO E CONSIDERAÇÕES DE DESEMPENHO

Neste capítulo é apresentada a validação funcional da arquitetura do processador matricial proposta neste trabalho. Esta validação restringe-se a uma simulação do comportamento esperado do módulo de controle através da implementação do interpretador descrito na seção 4.3. Pretende-se, com esta abordagem, comprovar os conceitos e a filosofia de operação que orientaram a definição da arquitetura proposta, explorando seu ponto fundamental: o processo de interpretação e geração de controle para a matriz de processadores.

A seguir, sobre características da arquitetura proposta, são feitas considerações a respeito dos principais fatores que determinam o desempenho e a escolha de aplicações para máquinas SIMD.

## 6.1 Validação funcional da arquitetura proposta

A arquitetura proposta para o processador matricial é composto por 3 partes: máquina hospedeira, unidade de controle, e matriz de processadores. Em função desta organização, quando da validação da arquitetura proposta, a atenção recai sobre a unidade de controle, pois tanto a máquina hospedeira como a matriz de processadores são dispositivos que isoladamente já tem seu funcionamento validado pelo seus respectivos fabricantes. Resta, então, garantir que a unidade de controle, e as interfaces entre estas 3 partes, sejam funcionalmente válidas.

A interface entre a máquina hospedeira e a unidade de controle é realizada através de sinais lógicos derivados do registrador de comandos (seção 4.2.3), que ativam, via circuitos digitais, os procedimentos de início de operação, carga de programas, inicialização do sistema e execução de "corner turning". Esta interface provê, ainda,

via registrador de status, a possibilidade de monitorar o estado operacional no qual o processador matricial encontra-se.

Já a interface entre a unidade de controle e a matriz de processadores é mais simples, sendo realizada pelos registradores de comando e endereço, por uma lógica de relógio, e pelo sinal GO (saída Global Output do GAPP). Nos registradores de comando e endereço são escritos pela unidade de controle o código da instrução a ser executada, e o endereço relativo a RAM local de cada elemento processador. A lógica de relógio funciona habilitando e desabilitando um ciclo de execução da matriz de processadores. A habilitação ocorre sempre após a escrita nos registradores de comando e endereço. O sinal GO, por sua vez, é enviado a unidade de controle possibilitando a execução de saltos condicionais no código interpretado, via instruções JMPGOHI e JMPGOLOW (anexo 4).

Em função destas características operacionais optou-se por validar funcionalmente a arquitetura proposta implementando o interpretador descrito na seção 4.3. Esta abordagem permite com que os conceitos envolvidos no módulo de controle, e o interfaceamento deste com a matriz de processadores sejam funcionalmente válidos. O simulador empregado implementa basicamente os procedimentos de carga de programas (seção 5.3), e o interpretador/gerador (seção 4.3). O simulador foi implementado empregando-se a linguagem C, e sua listagem é fornecida no anexo 6. Uma breve descrição da organização do simulador é dada a seguir.

A estrutura de dados empregada pelo simulador é bastante simples, sendo composta por 3 vetores de inteiros: **code_memory**, **stack_memory** e **variable_memory**. A cada um deste vetores está associado uma estrutura do interpretador (seção 4.3.1) no módulo de memória do processador matricial proposto. Os registradores internos (seção 4.3.2) são im-

plementados com o auxílio de 5 variáveis inteiras **auto_p**, **addr_p**, **pc**, **sp**, **arg_p** correspondendo, respectivamente aos registradores internos **auto pointer**, **address pointer**, **program counter**, **stack pointer** e **argument pointer**. Os registradores de comando e endereço, empregados no interfaceamento da matriz de processadores com a unidade de controle, também, são implementadas com 2 variáveis inteiras: **gapp_addr** e **gapp_inst**. É definido, ainda, a **struct ram_interna** que simula a existência da RAM interna do microprocessador 8031, necessária durante o procedimento de inicialização.

O primeiro passo executado é a chamada da rotina carregador, que implementa o programa carregador descrito na seção 5.3. As informações relativas a inicialização dos registradores internos do interpretador são lidos pelo carregador a partir do arquivo intermediário, e armazenados nas primeiras posições de área da pilha. Após a execução do carregador, as estruturas **code_memory** e **variable_memory** possuem, respectivamente, o código a ser interpretado e a inicialização das variáveis. A seguir as informações referentes a inicialização armazenadas na pilha são copiadas para a **struct ram_interna**, liberando a área de pilha.

Após a inicialização é executada a parte do simulador correspondente ao interpretador/gerador. Conforme apresentado na seção 4.3.3, o interpretador é basicamente um comando **switch** dentro de um **while**. A cada **case** do **switch** correspondem as ações de uma instrução do interpretador (anexo 4). O simulador fornece como saída os valores fornecidos pelo interpretador aos registradores de comando e endereço, compondo uma instrução para o GAPP.

Faz parte, ainda, do código do simulador a rotina a ser executada no procedimento de "corner turning". Como no sistema proposto esta rotina só faz sentido sendo executada pelo 8031, ela é descrita no código como comentário. Esta abordagem visa, de forma clara e organizada,

determinar algoritmicamente os passos a serem executados durante uma futura implementação em assembly 8031 do interpretador.

Para validar-se a arquitetura proposta foram realizados os procedimentos de compilação, carga e interpretação para alguns programas exemplos empregando-se o simulador acima descrito. Os programas exemplos selecionados foram: soma e subtração de dois números em complemento de dois, e a divisão e multiplicação de dois números em sinal magnitude. Este programas foram retirados da biblioteca de funções GAL distribuídas pela NCR, e seus fontes são fornecidos no anexo 7. Para cada um destes programas executados no simulador, obteve-se como saída o conjunto de instruções a serem enviadas a matriz de processadores. Estes resultados também são fornecidos no anexo 7.

Como resultado da simulação pode-se, para os programas exemplos, afirmar que: (a) o ambiente de programação proposto no capítulo 5 deste trabalho é funcional; (b) o interpretador/gerador, núcleo do módulo de controle, é operacional; e (c) a interface entre o módulo de controle e a matriz de processadores (saída do simulador) é executada de maneira adequada.

## 6.2 Considerações de desempenho

É objetivo desta seção analisar, exemplificando sobre estruturas da arquitetura proposta, a influência de cada um dos principais fatores que devem ser considerados na definição de projeto e de aplicações para máquinas SIMD. Estes fatores, a saber, são: tempo de E/S, complexidade de operações, comunicação interprocessadores e organização de dados em memória. Salienta-se, ainda, que esta análise não traduz o desempenho da arquitetura proposta. Esta abordagem fornece, apenas, uma idéia mais pragmática de como estes fatores atuam no tempo total gasto na avaliação de uma aplicação.

O tempo total gasto na execução de um algoritmo em uma arquitetura paralela pode ser expresso da seguinte forma:

$$T_t = T_e + T_c + T_{E/S}$$

onde,

Tt: tempo total de avaliação

$T_e$: tempo de execução

$T_c$: tempo de comunicação interprocessadores

$T_{E/S}$: tempo de E/S

O tempo de execução é o tempo dispendido na avaliação do algoritmo propriamente dito. O tempo gasto em primitivas de comunicação e de sincronização é denominado de tempo de comunicação interprocessadores. Por último, o tempo de E/S, é o tempo necessário a carga e a recuperação de dados. É objetivo, durante a escolha e implementação de algoritmos, a minimização destes tempos. A reordenação da sequência de operações, visando um aumento na quantidade de operações a serem executadas em paralelo, e a exploração adequada da organização de dados em memória são alguns

exemplos de como, atuando sobre $T_c$ e $T_e$, reduz-se o tempo total $T_t$. O tempo de E/S, por sua vez, é determinado pela arquitetura do sistema, devendo ser minimizado através do emprego de estruturas eficientes de hardware.

### 6.2.1 Tempo de E/S

O tempo gasto em operações de E/S pode representar uma parcela significativa no tempo total de avaliação de um algoritmo. Como consequência disto, durante o projeto de uma arquitetura deve-se prover meios eficientes para efetivar a entrada e saída de dados.

No caso específico dos processadores matriciais, o tamanho da matriz de processadores, além de uma eficiente estrutura de E/S, é um fator a ser considerado na redução do tempo de E/S. Esta influência é devido ao fato que, dependendo da aplicação, talvez haja a necessidade de dividir o problema em "n" subproblemas, menores o suficiente para serem processados na matriz. A divisão em subproblemas tráz como consequência um aumento no tempo de E/S, em função da necessidade de repetir "n" vezes os procedimentos de carga e recuperação de dados na matriz de processadores.

Considerando-se como exemplo o programa GAL, apresentado na figura 6.1 que implementa um algoritmo de mapeamento em níveis de cinza (seção 3.3.1), pode-se analisar de forma quantitativa a influência do tempo de E/S no tempo total de processamento em função do tamanho da matriz de processadores. Supõem-se, ainda, neste exemplo as seguintes condições: (a) a matriz de processadores é uma matriz quadrada, isto é, o número de processadores na dimensão "x" é igual ao número de processadores na dimensão "y"; (b) a largura dos dados a serem operados é de 8 bits, correspondendo a 256 níveis de cinza; e (c) o hardware responsável pelo "corner turning" é o descrito na seção 4.2.6.

```
/* Função:
        "thresh" compara cada pixel que compõem a imagem com um nível es-
        pecificado de "threshold" informando quais pixels que excedem es-
        te nível. Se um pixel possui um valor maior que o nível especifi-
        cado, a variável flag recebe o valor 1.

    Parametros:
                image   A  =  imagem de entrada (considera nro. sem sinal)
                image   F  =  este bit indica quando um pixel possui nivel
                              superior ao "threshold" (F=1).
                int     T  =  nivel de "threshold"

    tempo de execução:

        imagem com nivel de cinza em 4-bit          14 ciclos
                                      8-bit          26 ciclos
                                     16-bit          50 ciclos

*/

thresh (A, T, F)

image    A;                         /* imagem de entrada */
int      T;                         /* nível de threshold */
image    F;                         /* flag */
{
    int i;

    /* subtrai cada pixel do nível de threshold */

    c := 0;

    for (i = 0; i < size(A); i++)
    {
        /* Se bit "i" de T é 1, seta C igual a 1 e armazena BW anterior em NS */

        if(T & (1 << i))
            c := 1  ns := c;

        /* Se bit "i" de T é 0, seta C igual a 1 e aramzena BW anterior em NS */

        else
            c := 0  ns := c;

        /* carrega valor imagem em EW, threshold em NS, salva BW em C */

        A:i   ew := ram  ns := c  c := ns;
                  c := bw;
    }
    F:  ram := c;        /* armazena último BW (borrow) em F */
}
```

**figura 6.1 - Implementação do algoritmo de mapeamento em níveis de cinza**

A partir das considerações acima obtem-se a seguinte expressão para avaliar o tempo total de processamento:

$$t_t = t_e + t_{E/S}$$

$$t_{E/S} = (k+1).t_{CTLB}$$

$$t_{CTLB} = (3.B + 2 + N).M$$

onde,

$t_e$: tempo de execução do algoritmo

$t_{CTLB}$: tempo de execução para "corner turning"

M: número de linhas na matriz de processadores

N: número de colunas na matriz de processadores

k: número de vezes que o problema é subdividido

B: largura dos dados a serem operados

Para o caso exemplo, $t_e$ = 26 ciclos, B = 8, M e N iniciais iguais a 12. Convém salientar que o $t_{CTLB}$ é obtido da seguinte forma: a parcela (3.B + 2) corresponde ao tempo gasto, em ciclos, para a execução do algoritmo de "corner turning"; a parcela N corresponde ao tempo gasto pelo hospedeiro para inserir N dados no CTLB. Observa-se que a parcela N, na verdade, é uma função de N, definida pela capacidade de transferência de dados do hospedeiro para o CTLB. Para esta análise considerar-se-á a hipótese de que os dados são escritos no CTLB a uma taxa de um dado por ciclo de instrução do GAPP.

O algoritmo de mapeamento em níveis de cinza é executado sobre uma imagem de n x n pixels, a partir de uma matriz de processadores de dimensão 12 x 12. Repete-se, a seguir, este mesmo algoritmo para a mesma imagem, aumentando-se o tamanho da matriz de processadores em incrementos de 12 para cada dimensão, até que a dimensão da matriz de processadores seja maior ou igual que a da

imagem. Este procedimento força uma subdivisão em imagens menores enquanto a matriz de processadores for menor que a dimensão da imagem. A figura 6.2 mostra os tempos dispendidos na avaliação do algoritmo de mapeamento de níveis de cinza com imagens de tamanho 16x16, 32x32 e 64x64.

| dimensão | T(E/s) | T(e) |
|---|---|---|
| 12x12 | 8160 | 104 |
| 24x24 | 2016 | 26 |

(a) imagem 16x16

| dimensão | T(E/S) | T(e) |
|---|---|---|
| 12x12 | 39600 | 234 |
| 24x24 | 20160 | 104 |
| 36x36 | 4176 | 26 |

(b) imagem 32x32

| dimensão | T(E/S) | T(e) |
|---|---|---|
| 12x12 | 586080 | 936 |
| 24x24 | 102204 | 260 |
| 36x36 | 41760 | 104 |
| 48x48 | 55680 | 104 |
| 60x60 | 69600 | 104 |
| 72x72 | 14112 | 26 |

(c) imagem 64x64

**figura 6.2 - Tempos de execução T(E/S) + T(e) (em ciclos)**

A partir dos resultados levantados na figura 6.2 pode-se observar alguns pontos importantes. Primeiro, tomando-se para cada imagem o pior caso, matriz de processadores 12x12, e o melhor caso, matriz de processadores maior ou igual que a dimensão da imagem, verifica-se claramente a influência do "overhead" de execução causado pelo tempo de E/S, em função da subdivisão da imagem em tamanho me-

nores capazes de serem processados na matriz. Segundo, o aumento de processadores nem sempre reduz o tempo de total de avaliação podendo em alguns casos, inclusive, aumentá-lo (figura 6.2c). Esta característica decorre do mau aproveitamento do número de processadores em função do tamanho da matriz pelo tamanho do problema. Por exemplo, considerando-se uma imagem 64x64 em uma matriz de processadores organizada em 36x36 é necessário subdividir a imagem em 4 sub-imagens de dimensões 36x36, 36x28, 28x36 e 28x28. Ao empregar uma organização 48x48 a subdivisão é: 48x48, 48x16, 16x48, 16x16. Já para uma organização 60x60 a subdivisão obtida é: 60x60, 60x4, 4x60, e 4x4. Porém, a diminuição de linhas na sub-imagem não se traduz em redução de tempo, já que o número de processadores na matriz é fixo, como consequência o tempo gasto na organização 60x60 para introduzir uma imagem 60x4 é o mesmo que para introduzir-se a imagem 4x4. Neste caso, nota-se que dos 3600 processadores da matriz apenas 16 participam efetivamente do processamento; sendo que dos 60 processadores que compõem as linhas da matriz, apenas 4 linhas participam efetivamente da entrada de dados e do processamento, enquanto as outras 56 linhas participam apenas da entrada de dados, aumentando o tempo de comunicação interprocessadores sem representarem um aumento de computação.

### 6.2.2 Complexidade de operações

O tipo das operações a serem executadas no processador matricial influenciam no tempo total de processamento. Esta influência pode ser verificada analisando-se o tempo gasto (em ciclos) para execução das quatro operações aritméticas elementares em um elemento processador que compõem o GAPP.

A partir da figura 6.3 que fornece os tempos, em número de ciclos, de execução destas operações aritméticas, pode-se verificar que 3 fatores determinam o tempo total de

| | 4 bits | 8 bits | 16 bits |
|---|---|---|---|
| soma | 62 | 122 | 242 |
| subtração | 60 | 120 | 240 |
| divisão | 84 | 402 | 1758 |
| multiplicação | 47 | 271 | 1295 |

(a) Sinal magnitude

| | 4 bits | 8 bits | 16 bits |
|---|---|---|---|
| soma | 15 | 27 | 51 |
| subtração | 15 | 27 | 51 |
| divisão | 140 | 616 | 2188 |
| multiplicação | 160 | 504 | 1768 |

(b) Complemento de dois

| | 4 bits | 8 bits | 16 bits |
|---|---|---|---|
| soma | 13 | 25 | 49 |
| subtração | - | - | - |
| divisão | 138 | 516 | 1992 |
| multiplicação | 82 | 354 | 1474 |

(c) Sem sinal

**figura 6.3 - tempos de execução das operações aritméticas elementares**

execução em função da operação realizada: o tipo da operação propriamente dita, a representação numérica empregada, e a largura dos dados a serem operados. Observa-se que a execução das operações de divisão e multiplicação são as que requerem mais tempo de computação, se comparadas com as de soma e subtração. Esta característica demonstra que a inexistência de unidades especializadas para divisão e multiplicação no GAPP traduz-se em aumento no tempo de computação. Este, em função de outros fatores como, largu-

ra dos dados, número de processadores, quantidade destas operações, pode não justificar, ou até mesmo inviabilizar, o emprego do sistema para uma determinada aplicação.

O tipo da representação numérica necessária a aplicação também representa tempo gasto na avaliação da operação. Esta influência pode ser notada obervando-se o tempo gasto para uma mesma operação, com o mesmo número de bits de dado, para cada representação fornecida na figura 6.3

Por último, devido ao fato do GAPP ser um processador do tipo bit serial, a largura do dado a ser operado representa uma parcela significativa no tempo de avaliação. Esta influência pode ser claramente visualizada em cada linha da figura 6.3. Observa-se, ainda, a partir da implementação GAL das operações aritméticas elementares (anexo 7) que, enquanto as operações de soma e subtração possuem um crescimento linear, em função da largura do dado, as operações de multiplicação e divisão apresentam um comportamento quadrático.

Porém convém salientar que devido a característica SIMD apresentada pelo GAPP, estas operações são realizadas em paralelo em todos os elementos processadores, o que aumenta consideravelmente a performance do sistema. Por exemplo, o desempenho, no pior caso da figura 6.3, divisão de números de 16 bits em complemento de dois (2188 ciclos), supondo uma matriz de 72 elementos processadores, e um tempo de ciclo de 100ns, é de aproximadamente 3.29 MOPS, calculados da seguinte forma:

tempo da divisão: 2188x100ns = 21,88 us

nro divisão/segundo: 1/21,88us ≅ 45704 operações

total: 72 x 45704 ≅ 3.29 MOPS

### 6.2.3 Organização de dados em memória e comunicação entre processadores

Conforme apresentado na seção 1.2, as arquiteturas SIMD são classificadas, em função da sua organização de memória, em dois tipos: (a) com memória local a cada elemento processador, (b) com uma área de armazenamento compartilhada entre os elementos processadores. Os elementos processadores, por sua vez, são interligados através do emprego de algum tipo de rede de interconexão. Por uma questão de viabilidade de implementação física das arquiteturas SIMD estas estruturas de interconexão tendem a ser limitadas.

Esta limitação faz com que a organização de dados em memória, e a necessidade de comunicação interprocessadores tornem-se fatores importantes a serem considerados na escolha de uma aplicação. Por exemplo, uma organização adequada de dados pode, em função da aplicação, determinar um aumento na quantidade de operações a serem executadas. Por sua vez, a necessidade de troca de dados entre processadores (alta granularidade) se reflete em tempo de comunicação interprocessadores. Convém salientar que, em certas aplicações, o emprego de uma organização adequada de dados pode reduzir a necessidade de comunicação interprocessadores.

Como um exemplo de aplicação onde a organização de memória pode influenciar na redução do tempo de comunicação entre processadores, pode-se citar o algoritmo para cálculo da transformada de Sobel apresentado na seção 3.4. A avaliação do gradiente de Sobel é realizada computando-se a seguinte expressão:

$$G_s = MAX(|X_s|, |Y_s|) + 1/2\ MIN(|X_s|, |Y_s|)$$

A figura 6.4 apresenta o trecho de programa GAL que implementa e avalia a transformada de Sobel. Esta

implementação baseia-se em que os dados, ou seja, os pixels estejam mapeados em uma relação 1:1 com os elementos processadores fazendo com que a uma janela de imagem corresponda uma "janela" de processadores. Este mapeamento em função das características da arquitetura do GAPP permite que um processador receba o valor do pixel vizinho em um único passo de comunicação. Observa-se aqui, ainda, que o método empregado na avaliação de $X_s$ e $Y_s$, passos necessários para a avaliação da transformada de Sobel, exploram eficientemente o tipo de interconexão empregada pelo GAPP (mesh), e esta organização dos dados na memória.

```
image a:0:7;            /* area reservada para imagem de entrada */
image r:8:15;           /* area reservada para imagem de saida */
int t=60;               /* constante para efetuar operacao de Thresholding */
int i,j,n;              /* variaveis auxiliares /

main()
{
 image b:9,c:10;        /* areas de rascunho */
 image x:11;            /* valor absoluto de Xs */
 image y:11;            /* valor absoluto de Ys */
 image z:12;            /* magnitude */
 image flag:1;          /* flag resultado do nivel de threshold */

 image xF:1, xC:11, xD:11, xT:1; /*areas auxiliares*/
 {/* avaliacao de Ys */

  c:=0;
  n=size(a);
  i=0;
  /* avalia a primeira soma parcial */
  while (i<n) { a:i ew:=ram;
               a:i ns:=ram ew:=e;
               b:i ram:=sm c:=cy;
               i=i+1;}
  b:n ram:=sm c:=0;

  /* avalia a segunda soma parcial */
  i=0;
  while (i<n+1) {b:i ew:=ram;
                 b:i ns:=ram ew:=w;
                 c:i ram:=sm c:=cy;
                 i=i+1;}
  c:n+1 ram:=sm c:=0;

  /* avalia soma parciais */
  i=0;
  while (i<n+2) {c:i ns:=ram;
                 ns:=s;
                 c:i ns:ram ew:=ns;
                 ns:=n;
                 y:i ram:=sm c:=bw;
                 i=i+1;}
  y:n+2 ram:=sm c=0;
```

figura 6.4 - Implementação GAL do algoritmo de Sobel

```
/* Avaliacao de Xs*/
i=0;
/* avalia a primeira soma parcial */
while (i<n) { a:i ns:=ram;
               a:i ew:=ram ns:=s;
               b:i ram:=sm c:=cy;
               i=i+1;}
b:n ram:=sm c:=0;

/* avalia a segunda soma parcial */
i=0;
while (i<n+1) {b:i ns:=ram;
               b:i ew:=ram ns:=n;
               c:i ram:=sm c:=cy;
               i=i+1;}
c:n+1 ram:=sm c:=0;
/* avalia soma parciais */

i=0;
while (i<n+2) {c:i ew:=ram;
               ew:=w:=s;
               c:i ew:=ram ens:=ew;
               ew:=e;
               y:i ram:=sm c:=bw;
               i=i+1;}
y:n+2 ram:=sm c=0;


/*--- Obtencao dos valores absolutos de Xs E Ys ---*/

        /* Determinacao de Xs */

n=size(x);
x:n-1 c:=ram ns:=0;              /* bit MSB */
i=0;
while (i<n) {x:i ew:=ram;
             x:i ram:=sm;        /* sm= C xor EW */
             i=i+1;}

/* inverte valor se MSB=1 */

i=0;
while (i<n) {x:i ew:=ram;
             x:i ram:=sm c:=cy;
             i=i+1;}


        /* Determinacao de Ys */

n=size(y);
y:n-1 c:=ram ns:=0;              /* bit MSB */
i=0;
while (i<n) {y:i ew:=ram;
             y:i ram:=sm;        /* sm= C xor EW */
             i=i+1;}

/* inverte valor se MSB=1 */

i=0;
while (i<n) {y:i ew:=ram;
             y:i ram:=sm c:=cy;
             i=i+1;}
```

figura 6.4 - Implementação GAL do algoritmo de Sobel
(continuação)

```
/* determinacao dos valores max(Xs,Ys) e min(Xs,Ys) */

c:=0;
for(i=0;i(size(x)-1;i++) {x:=i ns:=ram;
                          y:=i ew:=ram;
                               c:=bw;}
xF:ram:=c;

/* determinacao min(Xs,Ys)=x*xF+Y*~xF ---) armazena em "d"*/

xF: ns:=ram;
for(i=0;i(size(x)-1;i++) {x:i  ew:=ram c:=0;
                               c:=cy;
                          xl:  ram:=c c:=0;
                          y:i  ew:=ram;
                               c:=bw;
                          xT:  ns:=c ew:=ram c:=1;
                          xF:  c:=cy ns:=ram;
                          xD:i ram:=c;}

/* determinacao max(Xs,Ys)=y*xF+x*~xF ---) armazena em "c"*/

xF: ns:=ram;
for(i=0;i(size(x)-1;i++) {x:i  ew:=ram c:=0;
                               c:=cy;
                          xT:  ram:=c c:=0;
                          y:i  ew:=ram;
                               c:=bw;
                          xT:  ns:=c ew:=ram c:=1;
                          xF:  c:=cy ns:=ram;
                          xC:i ram:=c;}

/* executa soma c + 1/2 d */

xD:0 c:=ram;
for(i=0;i(size(x)-2;i++) {xC:i   ns:=ram;
                          xD:i+1 ew:=ram;
                          z:i    ram:=sm c:=cy;}
xC:i  ns:=ram ew:=0;
z:i   ram:=sm c:cy;
z:i+1 ram:=c;


/* determinar bordas acima do nivel de threshold 60 */

thresh(z,t,flag);        /* subrotina de thresholding (fig. 6.1) */

/* a area de memoria "r" possui todos pontos acima do nivel */

          /* obtido a partir do "and" de "flag" com "z"*/

j=size(a);
flag: ns:=ram;
a:0 ew:=ram;
c:=0;
i=0;
while(i(j) {a:i+1 c:=cy ew=ram;
            r:i   ram:=c c:=0;
            i=i+1;}
}
```

**figura 6.4 - Implementação GAL do algoritmo de Sobel (continuação)**

O emprego de planos de memória (vide figura 3.8) é um outro exemplo de como a organização de dados pode facilitar a exploração do paralelismo. Observa-se que, após a avaliação de $X_S$ e $Y_S$, é necessário determinar-se as parcelas $MAX(|X_S|,|Y_S|)$, $MIN(|X_S|,|Y_S|)$ para cada ponto restringindo a um processamento local ao elemento processador. A definição de planos de memória para as variáveis empregadas nesta determinação, possibilita que com o envio de uma única instrução (endereço e comando) à matriz de processadores, todos os elementos processadores efetuem simultaneamente este cálculo.

# 7 CONCLUSÕES

A eficiência de uma arquitetura paralela na avaliação de uma determinada aplicação, depende fundamentalmente da combinação da estrutura do algoritmo utilizado com o tipo da arquitetura empregada. Esta combinação denomina-se de mapeamento. Hoje em dia, não existe uma metodologia que seja a um mesmo tempo formal, de uso prático e eficiente para obter-se este mapeamento. Como consequência, na maioria das vezes o mapeamento de um algoritmo a uma arquitetura é realizada de forma "ad hoc" respeitando-se certos compromissos. No caso específico de arquiteturas SIMD estes compromissos são: comunicação entre processadores, comunicação com o meio externo, complexidade de operações a serem executadas e organização de dados em memória.

A relação aplicação-algoritmo-arquitetura é evidenciada ao analisar-se uma área de aplicação cujas características adequam-se a arquiteturas SIMD; como a apresentada no capítulo 3 deste trabalho: tratamento digital de imagens no nível hierárquico de dados. As principais características que combinam esta aplicação com arquiteturas SIMD são a localidade e a simplicidade das operações realizadas.

Outro aspecto importante a ser considerado no uso de arquiteturas paralelas, é que as estruturas que exploram o paralelismo são vinculadas à máquina em que a aplicação será executada. Esta característica implica em que o programador deva ter o conhecimento da arquitetura da máquina empregada e levar isto em consideração quando da escrita do programa. Esta característica constitui-se em uma dificuldade adicional na elaboração de programas para máquinas paralelas.

Ainda, neste trabalho, como um caso prático de estudo e exemplo destas dificuldades propôs-se uma arquitetura do tipo SIMD. Foi proposto, então, um processador

matricial empregando-se o GAPP (Geometric Arithmetic Parallel Processor) que é um circuito integrado matricial disponível comercialmente. Na proposição da arquitetura foram levados em consideração os fatores de: (a) ser fisicamente implementável em nossa realidade atual, e (b) possibilitar o emprego da linguagem GAL (GAPP Algorithmic Language). A principal motivação para o emprego da linguagem GAL foi o ato de prover-se uma forma eficiente e prática de escrever programas para o sistema proposto. A desvantagem no uso da GAL reside no fato desta gerar código interpretado, o que causa um "overhead" no procedimento de avaliação.

Com base na arquitetura proposta foram desenvolvidos alguns exemplos de programas. Salientou-se a interação entre a arquitetura proposta e o meio externo via máquina hospedeira. Abordou-se, ainda, como os principais compromissos a serem considerados em arquiteturas SIMD atuavam sobre o processador matricial proposto. Notadamente observou-se a influência da complexidade das operações, e a organização e tamanho da matriz de processadores no tempo de execução de um algoritmo. Como referencial de tempo para esta análise utilizou-se o número total de ciclos gastos.

Como passos futuros deste trabalho vê-se um refinamento no projeto físico da arquitetura proposta, implementando-se, a nível de circuitos integrados, os módulos definidos no capítulo 4. Um outro passo é a implementação em assembly 8031 do software de controle (interpretador). A razão pela qual sugere-se a implementação em assembly é a otimização do código do interpretador, reduzindo assim o "overhead" no tempo de interpretação.

Finalmente, pensa-se ser a maior contribuição deste trabalho não a proposta de uma arquitetura paralela, mas sim a discussão sobre os compromissos envolvidos entre uma aplicação e a obtenção de arquiteturas paralelas para sua avaliação. Outro ponto importante, é a criação de uma

base para uma nova cultura de programação: paralelismo em máquinas SIMD.

# ANEXO 1

## Descrição do GAPP (Geometric Arithmetic Parallel Processor)

Este anexo fornece uma descrição do GAPP, apresentando aspectos relacionados a sua organização interna, princípio de operação, conjunto de instruções. Esta descrição foi obtida a partir de [NCR 85b] e [NCR 85a].

## DESCRIÇÃO DO GAPP (GEOMETRIC ARITHMETIC PARALLEL PROCESSOR)

O GAPP (Geometric Arithmetic Parallel Processor) é um processador paralelo, disponível comercialmente, fabricado pela NCR. Cada chip GAPP é composto por 72 elementos processadores, dispostos em uma matriz 12x6, que operam de forma síncrona executando uma mesma instrução sobre dados locais a estes processadores. O GAPP, internamente, é microprogramado horizontalmente.

Cada um dos elementos processadores possui linhas de comunicação bidirecionais que conectam um elemento processador com os processadores adjacentes nas direções norte, sul, leste e oeste ("vizinhança-4"), permitindo assim, a transferência de dados entre estes (figura A.1). A linha N/S conecta o processador aos processadores imediatamente acima e abaixo. A linha E/W conecta os processadores imediatamente a esquerda e a direita. A linha CMN/S conecta os processadores de forma análoga a linha N/S permitindo que haja transferência de dados nesta direção enquanto são realizados operações na ULA. O dado a ser transferido na linha CMN/S é armazenado em um registrador de trabalho (registrador CM).

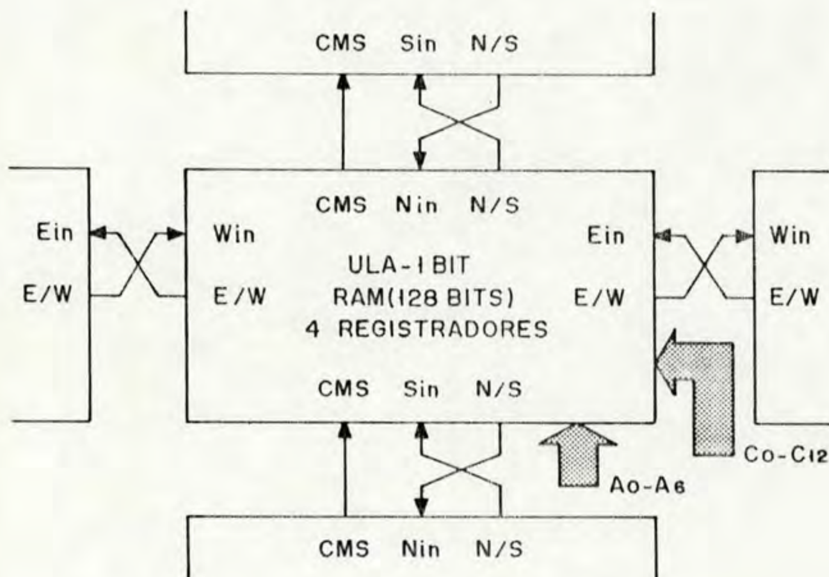



figura A.1 - Interconexão entre processadores

Os elementos processadores que compõem o GAPP são bastante simples, sendo constituídos por uma ULA de 1 bit, uma memória RAM 128X1, e um conjunto de 4 registradores internos de 1 bit. A figura A.2 mostra a arquitetura interna de cada elemento processador.

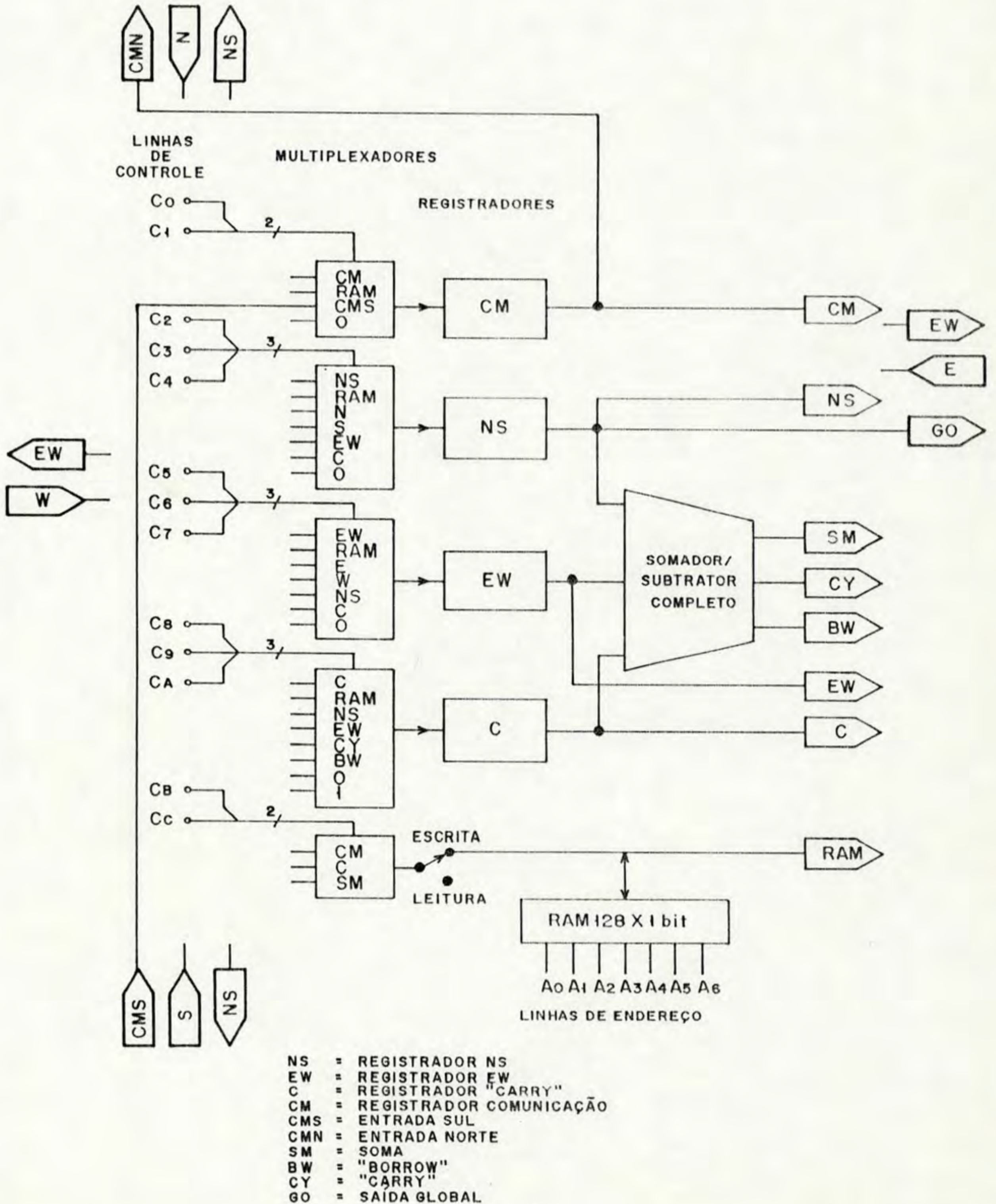



figura A.2 - Arquitetura Interna do Elemento Processador

UFRGS BIBLIOT

Os 4 registradores internos que compoêm o elemento processador são: CM, NS, EW e C. O registrador CM controla o fluxo de dados no barramento unidirecional referenciado como barramento CMN/S. O registrador NS tem como função principal o gerenciamento do fluxo de dados no barramento de dados bidirecional N/S. O Barramento N/S possibilita a comunicação entre os elementos processadores na direção norte e sul. O registrador EW, analogamente, manipula os dados no barramento bidirecional E/W. Este barramento permite aos elementos processadores a comunicação com os elementos processadores vizinhos na direção leste-oeste. Estes barramentos são todos de largura de 1 bit. O registrador C, por sua vez não é empregado para comunicação, sua função é estritamente local ao elemento processador, possuindo importância fundamental no funcionamento da ULA.

Cada registrador que compõe o elemento de processamento pode receber na sua entrada dados provenientes de multiplexadores que selecionam uma entrada de uma série de fontes. Por exemplo, o registrador C pode ser carregado com sua própia saída, a partir de uma posição da RAM, com o conteúdo dos registradores NS e EW, ou então receber as saídas CY (carry) e BW (borrow) da ULA, ou ainda receber os valores lógicos 0 e 1. Os demais de forma análoga podem receber entradas de uma variedade de fontes de acordo com o que fornece os multiplexadores a eles associados.

A ULA implementada nos elementos processadores é um somador/subtrator completo de suas entradas: o registrador NS, o registrador EW, e o registrador C. As saídas que a ULA fornece são três: soma (SUM), carry (CY), e borrow (BW). As saídas carry e borrow podem ser lidas diretamente do registrador C. Pode-se concluir que a funcao primária do registrador C é armazenar carry e borrow resultantes de operações na ULA. A ULA ao fornecer saídas de SM, CY, e BW permite a realização de operações aritméticas e lógicas das suas entradas de maneira bit serial. As operações lógicas executadas pela ULA dependem dos valores nos registradores C, EW e NS e são fornecidos na figura A.3.

**OPERAÇÕES ARITMÉTICAS**

OPERAÇÕES SOMADOR/SUBTRATOR

| ENTRADA | | | SAÍDA | | |
|---|---|---|---|---|---|
| NS | EW | C | SM | CY | BW |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |

**OPERAÇÕES LÓGICAS**

| OPERAÇÕES LÓGICAS | DESCRIÇÃO | CONDIÇÃO |
|---|---|---|
| INV | $SM = \overline{NS}$ | EW = 0, C = 1 |
| | $SM = \overline{EW}$ | NS = 0, C = 1 |
| | $SM = \overline{C}$ | NS = 0, EW = 1 |
| AND | $CY = NS \bullet EW$ | C = 0 |
| | $CY = EW \bullet C$ | NS = 0 |
| | $CY = NS \bullet C$ | EW = 0 |
| | $BW = \overline{NS} \bullet EW$ | C = 0 |
| OR | $CY = NS + EW$ | C = 1 |
| | $BW = \overline{NS} + EW$ | C = 1 |
| | $BW = EW + C$ | NS = 0 |
| XOR | $SM = NS \oplus C$ | EW = 0 |
| | $SM = NS \oplus EW$ | C = 0 |
| | $SM = EW \oplus C$ | NS = 0 |
| XNOR | $SM = \overline{NS \oplus EW}$ | C = 1 |

figura A.3 - Operações lógicas

Cada elemento de processamento possui uma RAM interna de 128 palavras de 1 bit. A RAM é de acesso direto exclusivo do elemento processador a si associado, elementos processadores vizinhos podem acessar indiretamente através dos registradores NS e EW empregando comunicação. A RAM é empregada essencialmente para armazenar palavras (dados) de forma a viabilizar tratamento de forma bit serial.

O GAPP utiliza um set de instruções composto por uma sequência de mnemonicos que controlam ações nos registradores e na RAM como mostrado na figura A.4. Observa-se nesta tabela que o GAPP é microprogramado horizontalmente o que implica em poder executar até cinco instruções simultaneamente, uma sobre cada um dos quatro registradores e uma sobre sua RAM. A microprogramação horizontal resulta no fato do GAPP poder executar aproximadamente 6000 instruções diferentes a partir de seu conjunto básico de instruções.

**CONJUNTO DE INSTRUÇÕES**

| | MNEMÔNICO | LINHAS DE CONTROLE | | | | | | | | | | | | | DESCRIÇÃO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $C_C$ | $C_B$ | $C_A$ | $C_9$ | $C_8$ | $C_7$ | $C_6$ | $C_5$ | $C_4$ | $C_3$ | $C_2$ | $C_1$ | $C_0$ | |
| CM | CM: = CM | | | | | | | | | | | | 0 | 0 | NOP |
| | CM: = RAM | | | | | | | | | | | | 0 | 1 | CARREGA CM EM RAM |
| | CM: = CMS | | | | | | | | | | | | 1 | 0 | TRANSFERE DE CMS PARA CM |
| | CM: = 0 | | | | | | | | | | | | 1 | 1 | CARREGA Ø PARA CM |
| NS | NS: = NS | | | | | | | | | 0 | 0 | 0 | | | NOP |
| | NS: = RAM | | | | | | | | | 0 | 0 | 1 | | | CARREGA NS EM RAM |
| | NS: = N | | | | | | | | | 0 | 1 | 0 | | | TRANSFERE DE N PARA NS |
| | NS: = S | | | | | | | | | 0 | 1 | 1 | | | TRANSFERE DE S PARA NS |
| | NS: = EW | | | | | | | | | 1 | 0 | 0 | | | TRANSFERE DE EW PARA NS |
| | NS: = C | | | | | | | | | 1 | 0 | 1 | | | TRANSFERE DE C PARA NS |
| | NS: = 0 | | | | | | | | | 1 | 1 | 0 | | | CARREGA Ø PARA NS |
| EW | EW: = EW | | | | | | 0 | 0 | 0 | | | | | | NOP |
| | EW: = RAM | | | | | | 0 | 0 | 1 | | | | | | CARREGA EW EM RAM |
| | EW: = E | | | | | | 0 | 1 | 0 | | | | | | TRANSFERE DE E PARA EW |
| | EW: = W | | | | | | 0 | 1 | 1 | | | | | | TRANSFERE DE W PARA EW |
| | EW: = NS | | | | | | 1 | 0 | 0 | | | | | | TRANSFERE DE NS PARA EW |
| | EW: = C | | | | | | 1 | 0 | 1 | | | | | | TRANSFERE DE C PARA EW |
| | EW: = 0 | | | | | | 1 | 1 | 0 | | | | | | CARREGA Ø PARA EW |
| C | C: = C | | | 0 | 0 | 0 | | | | | | | | | NOP |
| | C: = RAM | | | 0 | 0 | 1 | | | | | | | | | CARREGA C EM RAM |
| | C: = NS | | | 0 | 1 | 0 | | | | | | | | | TRANSFERE DE NS PARA C |
| | C: = EW | | | 0 | 1 | 1 | | | | | | | | | TRANSFERE DE EW PARA C |
| | C: = CY | | | 1 | 0 | 0 | | | | | | | | | CARREGA C PARA "CARRY" |
| | C: = BW | | | 1 | 0 | 1 | | | | | | | | | CARREGA C PARA "BORROW" |
| | C: = 0 | | | 1 | 1 | 0 | | | | | | | | | CARREGA Ø PARA C |
| | C: = 1 | | | 1 | 1 | 1 | | | | | | | | | CARREGA 1 PARA C |
| RAM | READ | 0 | 0 | | | | | | | | | | | | LÊ RAM |
| | RAM: = CM | 0 | 1 | | | | | | | | | | | | CARREGA RAM PARA CM |
| | RAM: = C | 1 | 0 | | | | | | | | | | | | CARREGA RAM PARA C |
| | RAM: = SM | 1 | 1 | | | | | | | | | | | | CARREGA RAM PARA SUM |

figura A.4 - Conjunto de Instruções

O controle do GAPP é efetuado a partir de 22 linhas externas. Sete linhas são utilizadas para endereçar a RAM interna dos elementos processadores. As instruções para o GAPP são enviadas através das linhas de controle C0-C12, as quais constituem a palavra do microprograma, que sendo internamente decodificada gera as ações de controle para a matriz de processadores. Uma linha de controle de status é provida pelo GAPP (GO), que é "ou" lógico do conteúdo de todos os valores contidos nos registradores NS. Por último tem-se o sinal de relógio, utilizado para a geração de ciclos para os processadores da matriz. O relógio em função do tipo de chip empregado pode ser de 5 ou de 10 MHz.

# ANEXO 2

## Linguagem GAL (GAPP Algorithm Language)

Este anexo descreve de forma suscinta a linguagem GAL. Como a linguagem GAL é uma linguagem "C-like", são enfatizados apenas os aspectos que diferem GAL de C. Esta descrição foi obtida a partir de [NCR 85a].

## LINGUAGEM GAL (GAPP Algorithmic Language)

### 1 INTRODUÇÃO

A linguagem GAL (GAPP Algorithm Language) é uma linguagem de alto nível, similar ao C, desenvolvida com o objetivo de tornar mais fácil e eficiente a criação de programas para o GAPP. Para atingir este objetivo a linguagem GAL provê meios para expressar e manipular de forma simples a RAM interna dos elementos processadores. Outra facilidade disponível é a geração de um código intermediário com notação pós-fixada a qual é adequada para a interpretação do código gerado por uma máquina de pilha. A vantagem do código intermediário é prover ao GAPP, com auxílio de estruturas de programação da linguagem C, meios para controlar o fluxo do programa como loops, saltos e subrotinas.

As diferenças existentes entre a linguagem GAL e a linguagem C são: (a) GAL suporta apenas variáveis do tipo inteiro, (b) as constantes em GAL só podem ser expressas em decimal, ou hexa, (c) a linguagem GAL provê acesso a instruções do GAPP de forma mnemônica, e (d) GAL provê um novo tipo de variável (image) voltada a utilização da RAM interna do elemento processador.

### 2. INCLUINDO INSTRUÇOES GAPP

Esta seção aborda dois tópicos: (a) como construir uma instrução GAPP, e (b) como expressar estas instruções em um programa escrito em GAL.

Cada instrução do GAPP é composta por até 6 campos, cada campo controlando operações sobre um dos seguintes registradores (NS, EW, C e CM), sobre a entrada da RAM, ou endereço da RAM. As operações válidas para cada campo são:

| | |
|---|---|
| campo RAM: | ram:=cm<br>ram:=c<br>ram:=sm ou ram:=plus |
| campo registrador C: | c:=nop ou c:=c<br>c:=ram<br>c:=ns<br>c:=ew<br>c:=cy ou c:=carry<br>c:=bw ou c:=borrow<br>c:=0<br>c:=1 |

campo registrador EW:    ew:=nop ou ew:=ew
                         ew:=ram
                         ew:=e ou ew:=east
                         ew:=w ou ew:=west
                         ew:=ns
                         ew:=c
                         ew:=0

campo registrador NS:    ns:=nop ou ns:=ns
                         ns:=ram
                         ns:=n ou ns:=north
                         ns:=s ou ns:=south
                         ns:=c
                         ns:=0

campo registrador CM:    cm:=nop ou cm:=cm
                         cm:=ram
                         cm:=cms
                         cm:=0

Uma instrução GAPP é composta de uma microinstrução de cada um dos seis campos descritos acima, podendo, assim, possuir de uma a seis microinstruções para o GAPP. Para formar uma instrução para o GAPP, deve-se escrever o menmônico da microinstrução desejada, separando-a por no mínimo um espaço em branco das demais microinstruções (mnêmonicos) que compoëm a instrução. A indicação de final de instrução é feita incluindo-se um ponto-e-vírgula (";"), após a lista de microinstruções que compoëm a instrução. Abaixo são listados alguns exemplos de instruções GAPP em GAL.

Exemplos:

```
c:=0 ns:=ew;
c:=cy ns:=0 ew:=c;
cm:=cms ns=north e:=bw
```

se uma microinstrução de campo não for incluída na lista de microinstruções que compoëm uma instrução GAPP, o compilador assume a execução de uma microinstrução nop para este campo.

## 3. VARIÁVEIS DO TIPO IMAGE

O emprego de variáveis do tipo **image** está relacionado com a utilização da memória RAM de cada elemento processador do GAPP. Em um programa GAL, o nome de uma variável do tipo **image** associada a uma expressão é utilizada para formar um endereço para RAM interna de cada elemento processador. O endereço inicial da variável **image** é definida a partir da declaração da variável, e a expressão, após avaliada, fornece um deslocamento ("offset") dentro da variável **image**. O endereço inicial e o deslocamento são somados para determinar qual endereço da RAM será associado. O formato de uma microinstrução que emprega o campo RAM é:

image nome:expressão

As partes **image** nome e expressão podem ser omitidas da microinstrução. Se o nome da variável é omitida, o endereço inicial assumido é zero, se a parcela expressão for omitida o deslocamento assumido é zero. Observa-se que expressão pode ser qualquer tipo de expressão válida para a linguagem C.

O endereço a ser acessado na RAM pode ser expres-

so na instrução GAPP de duas formas distintas. A primeira é colocar o endereço no início da linha que contém as micro-instruções que compoêm a instrução do GAPP. O endereço deve ser separado do menmônico seguinte por, no mínimo um branco. A segunda forma é incluir o campo de endereço depois da palavra reservada RAM. Caso o endereço da RAM não seja fornecido em uma instrução que a empregue, o último endereço utilizado será considerado (zero se nenhuma instrução previamente acessou a RAM). A seguir são fornecidos alguns exemplos do emprego de variáveis do tipo **image**.

Exemplos:

```
image a:4:7 ; /* variável "a" ocupa a RAM do en-
                 dereço 4 ao 7 inclusive */
c:=ram op1:(i+1) /* registrador C recebe o con -
                    teúdo de RAM(op1+i+1) */
ram(op2):=cm
```

Associado ás variáveis do tipo **image** existe uma função (SIZE) empregada para determinar o tamanho da variável **image**, i.é., a largura em bits que ocupa. O emprego da função SIZE é prática para controle de loops ao se manipular operações sobre variáveis image.

## 4 REGRAS DE ESCOPO PARA VARIÁVEIS

As variáveis são classificadas, de acordo com quais blocos de códigos podem acessá-las, em três classes: automáticas (**auto**), externas (**external**), e globais (**global**). Esta propriedade das váriaveis é conhecido como escopo.

As variáveis do tipo **auto** são aquelas variáveis do tipo **int**, ou **image**, que são declaradas dentro de um bloco, e existem somente dentro deste bloco. Da mesma forma, as variáveis declaradas no programa **main** existem somente dentro do programa principal, não sendo acessadas por subrotinas. As variáveis declaradas dentro de uma subrotina existem somente dentro da mesma, e não são conhecidas pelo **main**, nem pelas outras subrotinas. Caso haja necessidade do programa **main** e de uma subrotina compartilhar uma mesma variável, de forma a não empregar passagem de parâmetros, ou valores de retorno, é necessário utilizar variáveis do tipo **external** ou **global**.

As variáveis do tipo **external**, são aquelas variáveis declaradas fora da rotina que a emprega, e até mesmo em outro arquivos. As variáveis externas são referenciadas através da declaração **extern** dentro do bloco de código que a acessa. As variáveis do tipo **extern** podem também ser consideradas como do tipo global.

As variáveis declaradas fora do bloco de qualquer rotina são consideradas como variáveis do tipo **global**. As variáveis do tipo **global** são colocadas antes da palavra reserrvada **main**, ou entre o último "}" de uma subrotina, e o nome da próxima subrotina. Variáveis globais são acessadas por todos blocos de códigos que compoêm o programa.

# ANEXO 3

## Compiler GAL commands

Este anexo fornece as diretivas de compilação existentes no compilador GAL. Esta documentação foi obtida através da NCR Corporation, sendo parte integrante do apêndice D de [NCR 85a].

## COMPILER CG COMMANDS

The compiler is called from the command shell by the following command line:

CG [-A[-X]][-B][-T][-S][-L][-N num][-V] infiles [-O outfile]

where anything in square brackets ([]) is optional, either upper or lower case may be used. The infiles are the filenames of the inputs to the compiler (more than one file may be compiled at a time). These files must either be GAL source files or intermediate output files. If the filename has a .SIF extension, it is assumed to be an intermediate output file, otherwise it is assumed to be a GAL source file. The dash options, such as -S and -L, are used to control the operation of the compiler and have the following meanings.

-A

The second phase of CG will produce an ASCII output file which contains the GAPP instructions mnemonics and RAM address for the GAL program. This output file is suitable for use with the GAPP simulator (GAPSIM) as a "do" file. If this option is not given, the second phase will produce a binary output file which is the same format as the output file of the GAPP Assembler (GAPASM). If the -O option is also given, then the filename for the output file will be outfile. Otherwise, if the -A option is given the output filename will be OUT.G, and if the -A option is not given the output filenaem will be OUT.OBJ.

Note: if the -A option is not given, the -B and -X option are ignored.

-X

Controls the format of the RAM addresses in the output file produced by the -A option. When -X is given, the RAM addresses are printed as hexadecimal number (between 0 and 1F). Otherwise they are printed as decimal number (between 0 and 127).

Note: Both the -A and -X options are necessary when producing an output file to be used with GAPP simulator (GAPSIM).

-B

In a GAL source file, breakpoints may be included by using the program statement brkpoint. An example is

```
if (i==6) brkpoint;
```

which will produce a breakpoint when the int variables i is equal to 6. Breakpoints are understood by the GAPP Simulator (GAPSIM). When this option is given, the Simulator command brkpoint will be placed in the ASCII output file. If this option is not given, any brkpoint statement in the GAL source code will cause an error message during the second phase of the compiler.

-S

Only the first phase of the compiler is run. Any infiles which are not intermediate output files are compiled and the corresponding .SIF file is produced.
Note: If the -S option is given the -A,-X,-B,-L,-T,-N and -O options are ignored.

-L

Only the linking portion of the second phase of the compiler is run. All infiles which are not intermediate output files are compiled into .SIF files using the first phase of the compiler. Then all intermediate output files are linked together in memory. Instead of producing a final output file, howewer, another intermediate output files is produced which contains a linked version of all the intermediate output files. This linked intermediate output file is an exact copy of the intermediate output files linked together in memory which would normally be used to produce the final output file. If -O option is also given, the filename for this output is outfile. Otherwise the filename is OUT.LNK.
Note: using the linked intermediate output file as one of the infiles in a later run of the compiler is discouraged. This option is provided mainly for the PC development system.
Note: If the -L option is given the -A, -B, -X, -T, and -N options are ignored.

-N

As with any programming language, it is possible to get into an infinite loop. Since CG "unrolls" all loops, howewer, infinite loops can cause the second phase of the compiler to hang forever. To avoid the nasty possibilities this can imply, after 10000 GAPP instructions have been generated the user will be informed that there is a possible infinite loop, and will be asked whether to continue the compilation. The -N option changes the upper limit from 10000 to num.

-O

The output filename is outfile. Otherwise the default output filename is used (OUT.G, OUT.OBJ, or OUT.LNK depending on the -A and -L options).
Note: the following options are provided for use in-house by NCR and should have no importance to the user. They are documented here only to prevent confusion should they be accidently invoked by the user.

-T

Used by NCR in-house with the GAPP Tower Demonstration System. Automatically sets the -A option and appends a sequence of instructions necessary for correct operation on that system.

-V

Used by NCR in-house for debugging the compiler program. Turns on the "verbose" mode of the first phase.

# ANEXO 4

## The generator/interpreter model opcodes

Este anexo fornece uma descrição das instruções implementadas pela segunda fase do compilador GAL. Esta descrição apresenta as bases necessárias para um usuário que deseja empregar GAL em uma configuraçšo específica de hardware, criar (ou adaptar) a segunda fase do compilador para esta configuração. Esta documentação é parte integrante do apêndice D de [NCR 85a], obtida através da NCR Corporation.

## 5.4 THE OPCODES

Each of the opcodes stored in the code memory occupies one code memory address. Some opcodes indicate that there is more information in succeding code memory locations, but these other addresses hold data only (the opcodes are not modified by the data). Table 2 gives a list of opcode values vs. mnemonics (this would be useful as part of a header file for a generator/interpreter). Any opcode value not listed is either reserved for future use, or else is reserved to preserve compatibility with older versions of the compiler which use opcodes that have been obsoleted. The following sections describe the function of each opcode.

| mnemonic | value | mnemonic | value | mnemonic | value |
|---|---|---|---|---|---|
| ADDARG | 559 | ADDRAUTO | 537 | BLOCKEND | 526 |
| BLOCSTART | 540 | BRKPNT | 561 | CALL | 552 |
| CONDEND | 528 | ENTRY | 554 | GAPPADDR | 530 |
| GAPPINST | 529 | GETARG | 556 | GETAUTO | 541 |
| GETGLOB | 542 | JMP | 531 | JMPEQ | 532 |
| JMPGOHI | 533 | JMPGOLOW | 534 | LINK | 545 |
| OPBINMIN | 506 | OPBITAND | 518 | OPBITNOT | 504 |
| OPBITOR | 520 | OPBITXOR | 519 | OPCONST | 501 |
| OPDIV | 508 | OPEQ | 516 | OPGT | 513 |
| OPGTEQ | 516 | OPLOGAND | 521 | OPLOGNOT | 503 |
| OPLOGOR | 522 | OPLT | 512 | OPLTEQ | 514 |
| OPMOD | 509 | OPMULT | 507 | OPNOTEQ | 517 |
| OPPLUS | 505 | OPSHL | 510 | OPSHR | 511 |
| OPUMIN | 502 | POPSTK | 536 | PROGEND | 558 |
| PUTARG | 557 | PUTAUTO | 543 | PUTGLOB | 544 |
| RESUME | 555 | RETURN | 553 | REVSTK | 535 |
| START | 560 | STATEND | 525 | | |

Table 2 - Intermediate Output File Opcode Values

### 5.4.1 MAIN PROGRAM AND SUBROUTINE OPCODES

The following opcodes are used to implement the main routine and subroutine calling procedures.

**CALL** - Subroutine call. The arguments and the number of argument values have already been pushed onto the stack by previous opcodes. The code address of the first opcode of the subroutine is on the top of the stack, the amount to increment the auto pointer register is the next value down on the stack, and the amount to increment the address pointer is the second value down from the top the stack.

**ENTRY** - Subroutine entry point. This is the first opcode interpreted in a subroutine. The value in the stack pointer

is copied into the argument pointer register.

**PROGEND** - Program end. This is the opcode used to indicate that the end of the main program has been reached. Interpretation of the intermediate output files halts.

**START** - Program start. This is the first opcode of the main program. The initialization of the variable memory and internal registers mentioned in section 5.3 is assumed to have been done by the same initialization routine that loaded the opcodes into the code memory (see section 5.3). Any additional initialization or startup functions are performed when this opcode is encountered.

**RESUME** - Calling program resume. The amount to decrement the auto pointer register is on the top of the stack. The amount to decrement the address pointer register is the next value down on the stack. The second value from the top of the stack is the return value from the subroutine.

**RETURN** - Subroutine return. The return value is on the top of the stack. The return values is popped off the stack and written into the empty placeholder, the address of which is the value of the argument pointer minus 2. The value of the argument pointer is copied into the stack pointer register, and the return code address is popped off the new "top" of the stack into the program counter register.

### 5.4.2 READ/WRITE OPCODES

The following opcodes are used to read a data value from the variable or stack memory, or write a data value from the top of stack into the variable or stack memory structure.

**GETARG** - The data value to be read is one of the arguments to the subroutine, and is located in the stack memory. The argument offset of the data value to be read is on the top of the stack, where 0 is the offset of the first argument pushed onto the stack during the subroutine call. The number of arguments on the stack is read from the stack memory, the argument offset is popped off the stack, and the data value to be retrieved is read from the stack memory and pushed onto the stack. The address of the number of arguments is the value argument pointer register minus 4, and the address of the value to be read is the value of the argument pointer register minus 4, minus the number of arguments, plus the argument offset.

**PUTARG** - A data value is to be written into one of the arguments to the subroutine, which is located int the stack memory. The argument offset to be written is on the top of the stack where 0 is the offset of the first argument pushed onto the stack during the subroutine call. The value to be written is the next value down on the stack. The number of arguments on the stack is read from the stack memory, the argument offset is popped off the stack, and the data value to be written is read from the top of the stack (but not popped) and written into the stack memory address of the argument. The address of the number of arguments is the value of the argument pointer register minus 4, and the address of the argument to be written is the value of the argument pointer register minus 4, minus the number of arguments, plus the argument offset.

**GETAUTO** - The data value to be read is an auto int variable, and is located in the variable memory. The offset of the variable is on the top of the stack. The address of the variable is the value of the auto pointer register plus the offset. The offset is popped off of the stack, and the data value is read and pushed onto the stack.

**PUTAUTO** - The data value is to be written into an auto int variable which is located int the variable memory. The offset of the variable is on the top of the stack, and the

data value to be written is the next value down on the stack. The address of the variable is the value of the auto pointer register plus the offset. The offset is popped of the stack, and the data value is read from the top of the stack (but nop popped) and written into the variable memory address of the variable.

**GETGLOB** - The data value to be read is a global int or extern int variable and is located at the beginning of the variable memory. The absolute address of the variable is on the top of the stack. The address is popped off of the stack, and the data value is read from the variable memory address and pushed onto the stack.

**PUTGLOB** - The data value is to be written into a global int or extern int variable, which is located at the beginning of the variable memory. The absolute address of the variable is on the top of the stack, and the value to be written is the next value down on the stack. The address is popped off of the stack, and the data values is read from the top of the stack (but not popped) and written into the variable memory address.

### 5.4.3 STACK CONTROL OPCODE

The following opcodes are used to control the operation of the stack memory structure. "Pushing" a data value onto the top of the stack means that the stack pointer register is incremented by one and the data value is written into the stack at the new address in the stack pointer. "Popping" a data value off of the top the stack means that the data value is read off the stack memory address in the stack pointer, and the stack pointer register is them decremented by one.

**OPCONST** - A data value is pushed onto the top of the stack. The new data value is read from the code memory address int the program counter register and the program counter is incremented by one.

**REVSTK** - The data value on the top of the stack and the next data values down on the stack are exchanged.

**POPSTK** - The data value on the top of the stack is popped off of the stack.

### 5.4.4 OPERATOR OPCODES

The following opcodes perform arithmetic, boolean, and logical operations on data values which are on the top of the stack memory. Those operations which take two data values as operands pop two values off the top of the stack, and push the result onto the top of the stack. Those operations which take only one data value as an operand pop a value off the top of the stack and push the result onto the top of the stack. In the explanations below, the first operand is the data value on the top of the stack,and the second operand is the next data value down on the stack.

**OPBINMIN** - The result is the second operand minus the first operand.

**OPBITAND** - The result is the bitwise AND of the first and second operands.

**OPBITNOT** - The result is the bitwise inversion of the first operand.

**OPBITOR** - The result is the bitwise OR of the first and

second operands

**OPBITXOR** - The result is the bitwise exclusive OR of the first and second operands.

**OPDIV** - The result is the second operand divided by the first operand.

**OPEQ** - The result is a non-zero value if the first and second operands are equal. Otherwise the result is zero.

**OPGT** - The result is a non-zero values if the second operator is greater than the first operand. Otherwise the result is zero.

**OPGTEQ** - The result is a non-zero values if the second operator is greater than or equal to the first operand. Otherwise the result is zero.

**OPLOGAND** - The result is a non-zero value if the first and second operands are both non-zero. Otherwise the result is zero.

**OPLOGNOT** - The result is a non-zero value if the first operand is non-zero. Otherwise the result is zero.

**OPLOGOR** - The result is a non-zero value if either the first or second operand are non-zero. Otherwise the result is zero.

**OPLT** - The result is a non-zero value if the second operand is less than the first operand. Otherwise the result is zero.

**OPLTEQ** - The result ia a non-zero value if the second operand is less than or equal to the first operand. Otherwise the result is zero.

**OPMOD** - The result is the remainder of the second operand divided by the first operand.

**OPMULT** - The result is the multiplication of the first and second operands. Note that the result is the same size (number of bits) as the two operands.

**OPNOTEQ** - The result is a non-zero values if the first and second operands are not equal. Otherwise the result is zero.

**OPPLUS** - The result is the addition of the first and second operands.

**OPSHL** - The result is the second operand shifted left by the number of bit positions given in the first operand. Bits shifted out of the most significant bit position are lost. Bits shifted into the least significant bit position are zero.

**OPSHR** - The result is the second operand shifted right by the number of bit positions given in the first operand. Bits shifted into the most significant bit position may be either the same as the previous most significant bit (for arithmetic fill) or zero (for logical fill). Bits shifted out the least significant bit position are lost.

**OPUNMIN** - The result is the arithmetic negative of the first operand.

### 5.4.5 GAPP INTERFACE CODES

The following opcodes are used to generate GAPP instructions an RAM address.

**ADDRARG** - The value on the top of the stack is the argument offset of an image variable, which is to be converted to the starting GAPP RAM address of the image variable. The number of arguments on the stack is read from the stack memory, the argument offset is popped off of the stack, and the starting address is read from the stack memory and pushed onto the stack. The address of the number of arguments is the value of the argument pointer register minus 4, and the address of the value to be read is the value of the argument pointer register minus 4, minus the number or arguments, plus the argument offset.
Note: the ADDRARG opcode function exactly the same as the GETARG opcode and will be removed from future version of CG.

**ADDRAUTO** - The value on the top of the stack is the offset of an auto image variable, which is to be converted to the starting GAPP RAM address of the image variable. The offset is popped off of the stack, and the starting address is pushed onto the stack. The starting addres is the offset plus the values of the address pointer register.

**GAPPADDR** - The value on the top of the stack is the RAM address for the next GAPP instruction. The address is popped off on the stack and saved until the next GAPPINST opcode is encountered, at which time the address and instructions are written to the output file.

**GAPPINST** - The value on the top of the stack is a GAPP instruction. The instruction is popped off of the stack, and is written to the output file, along with the address from the last GAPPADDR opcode.

### 5.4.6 JUMP OPCODES

The following opcode are used for program flow control.

**JMP** - The value on the top of the stack is the code address of the next opcode to be interpreted. The address is popped of the stack, and written into the program counter register.

**JMPEQ** - The value on the top the stack is the code address of the next opcode to be interpreted if the next value down on the stack is zero. The address and the value are popped off of the stack, and if the value is zero the address is written into the program counter register. Otherwise, the program counter is left unchanged.

**JMPGOHI** - The value on the top of the stack is the code address of the next opcode to be interpreted if the GAPP global output (GO) pin is high. The address is popped off of the stack, and the status of the global output (GO) is checked. If the global output is high, the address is written into the program counter register, otherwise the program counter is left unchanged.

**JMPGOLOW** - The value on the top of the stack is the code address of the next opcode to be interpreted if the GAPP global output (GO) pin is low. The address is popped off of the stack, and the status of the global output (GO) is checked. If the global output is low, the address is written into the program counter register, otherwise the program counter is left unchanged.

NOTE: the JMPGOHI and JMPGOLOW opcodes are supported by the generator/interpreter which is part of the PC Development System, but will cause an error if they are encountered by the generator/interpreter in the second phase of the compiler CG.

### 5.4.7 THE LINK OPCODE

Because a GAL program can be distributed across several files, it is not possible to resolve all references to variables and subroutines during the first phase of the compiler. Therefore, a method of postponing the reference is used, which makes use of the LINK opcode. The LINK opcode is not a true opcode, since it is intercepted by the linking part on the second phase and replaced whit an OPCONST opcode. The LINK opcode is followed by ten words in the intermediate output file, which are replaced by a single word which becomes the value associated with the OPCONST code. The first word following the LINK opcode is the offset, the second word is the label number, and the last eight words are a symbol name, with one character of the symbol name per word (in ASCII format). Each of the three entries following the LINK opcode (the offset, the label number, and the symbol name) are translated into a value, all of which are added together to obtain the final value associated with the OPCONST opcode.

The offset value is added directly into a final value.

If the label number is 4 or greater, then the value associate with that label number is read from the the label table in the intermediate output file and added into the final value. Label numbers 0 through 3 have special meaning as described below.

Label number 0
Ignore the label number entry. Add nothing into the final value.

Label number 1
Translate this label number into a value equal to the amount of variable memory space taken up by intermediate output files already linked together. This is the sum of the initialization data count entry in the intermediate output file headers. If this is the only intermediate output file being linked, or is the first file in the list of files being linked, then the value will be zero. Add this value into the final value.

Label number 2
Translate this label number into a value equal to the starting address of the image variable given in the symbol name entry. The symbol tables of each of the intermediate output files being linked are examined until the symbol name is found. Then the starting address entry is read and added into the final value.

Label number 3
Translate this label number into a value equal to the size (number of bits) of the image variable given in the symbol name entry. The symbol tables of each of the intermediate output files being linked are examined until the symbol name is found. Then the size entry is read and added into the final value.

If the first word of the symbol name entry is zero, then the entry is ignored and nothing is added into the final value. Otherwise, the symbol tables of each of the intermediate output files being linked are examined until the symbol name is found. Then the offset of the symbol is read and added into the final value (unless the label entry is 2 or 3, in which case the starting address entry or size entry is used).

5.4.8 DIRECTORY OPCODES

The following opcodes are provided so that a generator/interpreter may have information about what portion of a GAL source file is currently being processed. This may be useful for debbuging and optimization.

**BLOCKEND** - The previous opcode was the last opcode of a block in the GAL source code. A block is composed of any GAL program statements enclosed in brackets ({}). The compiler assigns a unique block number to each block for the purpose of debbuging, and the number of the block which surrounds the block just exited is in the next code address. If this is the outermost block of a main routine or subroutine, then the surrounding block number will be zero.

**BLOCKSTART** - The next opcode is the first opcode of a new block in the GAL source code. A block is composed of any GAL program statements enclosed in brackets ({}). The compiler CG assigns a unique block number to each block for the purposes of debugging, and the number of the new block follows in the next code address.

Note: The generator/interpreter in the PC development system uses the BLOCKEND and BLOCKSTART opcodes to keep track of which variables are active (auto variables are only active inside the block in which they are declared, while global adn extern variables are always active). Any active image variable may be uploaded using the "upload RAM" option in the debug menu.

**BRKPOINT** - This opcode indicates that a brkpoint statement appeared in the GAL source code at this point.
Note: The generator/interpreter in the compiler CG will produce the simulator command brkpoint whenever the BRKPOINT opcode is encountered. The generator/interpreter in the PC development system will suspend interpretation of the intermediate output file whenever the BRKPOINT opcode is encountered. The user may use any of the commands in the debug menu, and may resume interpretation of the intermediate output file by using the "G GAPP instr", "B Breakpoint", or "F Finish program" options in the debug menu.

**CONDEND** - The previous opcode was the end of a complex conditional statement. The conditional statement may be composed of more than one statement (each of which is marked by a STATEND opcode), and even one or more blocks (marked by BLOCKSTART and BLOCKEND opcodes).

**STATEND** - The previous opcode was the end of a GAL program statement.

# ANEXO 5

## The intermediate output file format

Este anexo fornece a descrição do formato do arquivo intermediário (.SIF) gerado pela primeira fase do compilador GAL. Este anexo contém informações necessárias para realizar a adaptação da segunda fase do compilador para uma configuração de hardware específica do usuário. Esta documentação é parte integrante do apêndice D de [NCR 85a], obtida através da NCR Corporation.

# 6.THE INTERMEDIATE OUTPUT FILE FORMAT

The intermediate output file contains binary information produced by the first phase of the compiler program CG. The file contains not only the opcodes interpreted by the genrator/interpreter, but also contains information about variable and subroutine names (the symbol table), initialization information (the initialization data table), and information about labels used for subroutines calls and jump destinations (the label table). Each of these table is found in the intermediate output file in the order in which are discussed below. Also discussed is how each table of the linked intermediate output file, which results from linking several intermediate files together is obtained.

## 6.1 THE FILE HEADER

The file header serves as a directory of the rest of the information in the intermediate output file, and also holds important information for the linker. The header is made up several entries, each of which takes up one word int the file, and which are arranged in the order in which they area discussed below.

**symbol table count** - contains the number of entries in the symbol table of the intermediate output file. Each symbol table entry is made up of eight bytes followed by seven words (see section 6.2).

**label table count** - contains the number of entries in the table. Each label table entry is made up of two words (see section 6.4).

**initialization data count** - contains the number of entries in the initialization data table (see section 6.3). Each initialization data table entry is made up of one word. Since each initialization data entry is written into one variable memory address, the initialization data count is also the number of variable memory addresses used by this file for global int and extern int variables.

**actual code count** - contains the actual number of entries in the code table. Each entry in the code table consists of one word, and is either an opcode to be loaded into the code memory, a value associated with an OPCONST opcode, or part of a list of ten values associated with a LINK opcode.

**linked code count** - contains the number of entries in the code table minus nine entries for every LINK opcode in the code table. This is the number of code memory addresses which will be used by the intermediate output file after the link opcodes have all been converted to OPCONST opcodes by the linker.

**absolute RAM count** - contains the number of GAPP RAM addresses used by global image and extern image variables which are declared using the absolute declaration form. The absolute declaration form is

image name: start_address: end_address;

since no relatively declared image variable are allowed to occupy space dedicated to a absolutely declared image variable, the absolute RAM count entry is the maximum of the end_address of all absolutely declared image variables in a GAL source file.

**relative RAM count** - contais the number of GAPP RAM addresses used by global image and extern image variables which are declared using the relative declaration form. The relative declaration form is

image name:number_bit;

**block count** - contains the number of blocks used by this intermediate output file.

**label number** - contains the number of the highest label used by this intermediate output file plus one. Note that because not all label number are used, this entry is not the same as the label table count.

**executable flag** - set to one if this intermediate output file contains the main routine, zero otherwise.

**executable offset** - if the executable flag is one, contains the offset within the code table where the main routine begins. Note that this offset counts all LINK opcodes and associated values as two entries (i.e. this offset is valid only after the LINK opcdes have been resolved into OPCONST opcode).

Each entry of the linked header is obtained by summing the corresponding entries from the header of each intermediate output file being linked. The only exceptions are the absolute RAM count, the executable flag, and the executable offset entries.

The absolute RAM count entry of the linked header is the maximum of the absolute RAM count entries of all the intermediate output files being linked.

The executable flag entry of the linked header is set to one if any of the intermediate output files being linked have the executable flag entry set to one. It is an error for more than one intermediate output file to have the executable flag set to one. The executable offset entry is set when the intermediate output file which has the executable flag set to one is encountered and is left alone at all other times. When this file is encountered, the executable offset entry of the linked file is set to the executable offset entry of the file plus the sum of the actual code count entries of all intermediate output files linked prior to this file. Thus the executable offset is the actual offset of the main routine within the linked code table (see section 6.5).

## 6.2 THE SYMBOL TABLE

The symbol table contains informations about all image and int variable names, and all subroutine names used in the GAL program source file. The table is made up of the number of entries given in the symbol table count entry of the file header. Each symbol table entry consists of eight bytes and seven words, and is composed of the following entries, arranged in the order in which are discussed.

**symbol name** - consists of eight bytes, which is the first eight characters of the name of the variable or subroutine. The first byte of the entry is the first character of the name, the second byte is the second character, etc.

note: all of the remaining symbol entries consist of one word each.

**type** - indicates the meaning of a value assigned to a variable or the meaning of the value the subroutine returns. Each of the following entry values has the meaning described below. Any other type entry value is illegal.

- 0 - int variable or subroutine returning int value
- 1 - absolutely declared image variable
- 2 - relatively declared image variable

note: see section 6.1 for absolute and relative declarations form.

**class** - indicates where the variable is stored, or where the subroutine is located. Each of following entry value has the meaning described below. Any other class entry value is illegal.

- 0 - auto variable (i.e. declared locally in the main routine or in a subroutine).
- 1 - global variable or subroutine
- 2 - variable which is an argument to a subroutine
- 3 - extern variable or subroutine
- 4 - undefined variable or subroutine. This class code is only used internally by the compiler and should not be seen in an intermediate output file.

**function flag** - set to one if the symbol belongs to a subroutine. The flag is zero otherwise. If this flag is set to one, the type entry must be 1, and the class entry must be 1 or 3.

**offset** - only valid for symbols with type entry 1, and class entry 0,1, or 2, or for symbols with type entry 2 or 3, and class entry 2. Its meaning depends on the type and class entry values. Each of the following type and class entry values results in a meaning for the offset entry as described below. The offset entry is undefined for any other type and class entry values.

type 1, class:

0 - the offset entry is the offset within the main routines or subroutines space for auto int variables in the variable memory. This is the offset value used with the GETAUTO and PUTAUTO opcodes.

1 - if the function flag entry is not set to one, the offset entry is the absolute variable memory addres of the int variable. This is the offset values used with the GETGLOB and PUTGLOB opcodes. If the function flag entry is set to one, the offset entry is the offset within the code table of the beginning of the subroutine.

2 - the offset entry is the argumet number of the int subroutine argumet. This is the offset value used with the GETARG and PUTARG opcodes.

type 2 or 3, class:

2 - the offset entry is the argumet number of the starting address of the image subroutine argument. Recall that an image variable name as a subroutine argument actually results in two subroutine argument pushed onto the stack. This is the offset value used with the ADDRARG opcode.

**starting address** - only valid for symbols with type 2 or 3, and its remaning depends on the class entry. Each of the following class entry values results in a meaning for the starting address entry as described below. The starting address entry is undefined for any other class entry value.

- 0 - the starting address entry is the starting address offset within the main routines or subroutines auto image space in the GAPP RAM. This is the offset used with the ADDRAUTO opcode.

1 - the starting address entry is the absolute GAPP RAM starting address of the global image variable. This the offset used with the GAPPADDR opcode.

**size** - only valid for symbols with type entry 2 or 3. This entry gives the number of bits which the image variable is declared to have.

**block number** - gives the unique block number assigned by the compiler in which this variable or subroutine was declared. If the symbol has class entry 1, then the block number entry will be zero.

As each intermediate output file is linked, the symbols from its symbol table are installed into the linked symbol tables. Symbols which have class 1 (global) are checked to see that they do not conflict with any other global symbols. Symbols already in the linked symbol table which have class 3 (extern) or (4 undefined) may be replaced by global symbols with the same name as they are encountered. At the end of the linking procedure, it is an error for any extern or undefined symbols to be left in the linked symbol table.

As part of installation procedure, the entries of the symbol table in the intermediate output file are modified before installation into the linked symbol table. As an example, consider a symbol which has type entry 0 (int) and class entry 1 (global). This variable is to be stored in a variable memory address, the value of which is held in its offset entry. Howewer, there may be (and probably are) other global int variables with the same offset entry values in other intermediate output file, so the offset entry of the symbol must be modified in order not to conflict with other variables. This is a case of shared resource, parts of which must be allocated to each intermediate output file. A complete list of the shared resources, the associated entry, and the action taken to modify the entry is given below.

**code memory addresses**
For all symbols which have class 1 (global), type 0 (int), and the function flag set to one, the offset entry is increased by the sum of the linked code count header entries of all previously linked intermediate output files.

**variable memory addresses**
For all symbols which have class 1 (global), type 0 (int), and the function flag not set to one, the offset entry is increased by the sum of the initialization data count header entries of all previously linked intermediate output files.

**GAPP RAM addresses**
For all symbols which have class 1 (global), and type 2 (relatively declared image) the starting addres entry is increased by the sum of the relative RAM count header entries of all previously linked intermediate output files.

Note: relatively declared global image variables are not allowed to conflict with (i.e. share the same GAPP RAM addresses eith) absolutely declared global image variables. Howewer, there is no way to know how much GAPP RAM space is taken up by absolutely declared global image variables until all intermediate files have been linked. Rather than making a second pass through the symbol table to modify all starting address entries for relatively declared global image variables, there is a convention that all references to the starting address of a relatively declared global image variable shall be though the use of a LINK opcode, using the special label value 2. This way the starting address entries are modified when the LINK is resolved after the symbol table is installed.

note: it may seem intuitive that the starting address

entries of absolutely declared global image variables should also be modified. Howewer, these image variables are allowed to conflict (with other absolutely declared image variables only), and so no partitioning of the resource is needed.

Compiler-assigned unique block numbers:
two blocks from different source files should not have the same block number in the linked file. For all symbols which do not have class 1 (global), 3 (extern), or 4 (undefined) the block number entry is increased by the sum of the block count header entries of all the previously linked intermediate output files.

## 6.3 THE INITIALIZATION DATA TABLE

The initialization data table contains data which is to be loaded into variable memory addresses prior to interpretion of the intermediate output file. The number of entries is given in the initialization data count header entry, and each entry takes one word. The first entry is to be loaded into variable memory address zero, the second is to be loaded into address one, etc. Howewer, since the variable memory addresses are shared resources (see section 6.2), the addresses into which the data is to be loaded must be increased by the sum of the initialization data count header entries of all previously linked intermediate output files.

## 6.4 THE LABEL TABLE

The label table records the matching between values and the label numbers used with LINK opcodes. The number of entries is given in the label table count header entry, and each entry takes two words. The first word of the entry is the label number, and the second word is the associated value.

The label number, like the others items discussed in the section 6.2 are a shared resource. Therefore as the labels from an intermediate output file are being installed into the linked label, the label number of each entry is increased by the sum of the label number header entries of the previously linked intermediate output files.

## 6.5 THE CODE TABLE

The code table holds the opcodes and LINK entries of the intermediate output file. The number of entries in the code table is given in the actual code count header entry, and each entry consists of one word. After removal of the LINK opcodes and associated values, the number of entries will be the value in the linked code count header entry.

There is a complication involving the removal of LINK opcodes. The information from the symbol table of all the intermediate output files being linked is needed to ensure that all LINK entries can be resolved. To accomplish this, the linking procedure is done in two phases. On the first phase, the header, symbol table, label table, and initialization data table are linked from each intermediate output file, and the code table is skipped over. On the second phase, the header is re-linked (to ensure the availability of the same information about the sum of header entries of previously linked intermediate output files), the label table, symbol table, and initialization data table are all skipped over, and the code table is linked.

The entries in the code table are to be loaded

into the code memory, with the first entry in the table being loaded into code memory address zero, the next entry loaded into address one, etc. The code memory addresses, as discussed in section 6.2, are a shared resource. Therefore, before each code table entry of an intermediate output file is loaded into code memory, the code memory destination address is increased by the sum of the linked code count header entries of the previously linked intermediate output files.

As part of the linking procedure, most opcodes encountered from the code table are simply loaded directly into the code memory. Certain opcodes are intercepted, howewer, and cause special action to be taken.

**OPCONST** - the OPCONST opcode is loaded directly into code memory. Howewer, the value following the opcode in the code table may be any value, including the same value as an intercepted opcode. This value is loaded directly into code memory following the opcode without any interceptation.

**LINK** - An OPCONST opcode is loaded into code memory. The next ten values form the code table are translated into a single values, which is loaded into code memory following the opcode.

**BLOCKSTART** and **BLOCKEND** - The values in the code table following these opcodes are compiler-assigned unique block numbers, which are shared resources as discussed in section 6.2. The opcodes are loaded into code memory. The block numbers are increased by the sum of the block count header entries of the previously linked intermediate output files, and then loaded into code memory following the opcode.

## 6.6 THE LINKED INTERMEDIATE OUTPUT FILE

At the end of the linking procedure, there is a linked header, a linked symbol table, and a linked label table stored in general purpose memory. The initialization data table and code tables from all the intermediate output files are linked and stored in variable memory and code memory, respectively. At this point, the generator/interpreter is ready to begin interpreting the opcodes to generate the GAPP instructions and RAM addresses for the final output file.

Howewer, if the -L option is used to request a linked output file, the following will occur. The linked header will be written into the linked intermediate output file with some minor modifications. All LINK opcodes will have been removed, so the actual code count entry and the linked code count entry will be the same. There is no need for the label table information, so the label table count entry will be zero. The symbol table will be written to the linked intermediate output file with no modification. The linked initialization table will be written to the linked intermediate output file from the variable memory. The linked label table will not be written to the linked intermediate output file (the information is no longer needed). Finally the linked code table, with all LINK opcodes resolved, will be written to the linked intermediate output file from the code memory.

# ANEXO 6

## Listagem do simulador

Este anexo fornece a listagem do interpretador-gerador desenvolvido como parte integrante deste trabalho. Este programa (simulador) implementa a segunda fase do compilador GAL para a arquitetura proposta no capítulo 4. Sua principal função é ler o arquivo intermediário (.SIF) obtido pela primeira fase do compilador GAL, e gerar os comandos para a matriz de processadores (GAPP).

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <dir.h>
#include <fcntl.h>
#include <alloc.h>
#include <io.h>
#include <string.h>
#include <stdlib.h>
#include <stat.h>
```

```
/*
------------------------------------------------------------------------------

                    INTERPRETADOR/GERADOR - VERSAO 1.00

        Este programa implementa o interpretador apresentado na secao 4.3. E
composto por 2 modulos distintos: carregador e o interpretador/gerador.
        O carregador executa a leitura do arquivo intermediario gerado pelo
compilador GAL, recuperando a partir do "header" as informacoes de formato e de
tamanho deste. Estas informacoes sao necessarias para o procedimento de
interpretacao do arquivo.
        O interpretador/gerador, a partir do arquivo intermediario lido pelo
carregador, executa um procedimento de interpretacao implementando as funcoes de
controle da segunda fase do compilador GAL. Como saida fornece a sequencia de
comandos a serem executados pela matriz de processadores GAPP.


        Autor: Alexandre da Silva Carissimi
        Linguagem: C (compilador Turbo C 1.5 da Borland)
        Data ultima alteracao: 17/04/89
        Versao: 1.00

------------------------------------------------------------------------------
*/
```

```
/*
-------------------------------------------------------------------------------
|                                                                             |
|                    Opcodes do Gerador/Interpretador                         |
|                                                                             |
-------------------------------------------------------------------------------
*/

#define        OPCONST        501
#define        OPUNMIN        502
#define        OPLOGNOT       503
#define        OPBITNOT       504
#define        OPPLUS         505
#define        OPBINMIN       506
#define        OPMULT         507
#define        OPDIV          508
#define        OPMOD          509
#define        OPSHL          510
#define        OPSHR          511
#define        OPLT           512
#define        OPGT           513
#define        OPLTEQ         514
#define        OPGTEQ         515
#define        OPEQ           516
#define        OPNOTEQ        517
#define        OPBITAND       518
#define        OPBITXOR       519
#define        OPBITOR        520
#define        OPLOGAND       521
#define        OPLOGOR        522
#define        STATEND        525
#define        BLOCKEND       526
#define        CONDEND        528
#define        GAPPINST       529
#define        GAPPADDR       530
#define        JMP            531
#define        JMPEQ          532
#define        JMPGOHI        533
#define        JMPGOLOW       534
#define        REVSTK         535
#define        POPSTK         536
#define        ADDRAUTO       537
#define        BLOCKSTART     540
#define        GETAUTO        541
#define        GETGLOB        542
#define        PUTAUTO        543
#define        PUTGLOB        544
#define        LINK           545
#define        CALL           552
#define        RETURN         553
#define        ENTRY          554
#define        RESUME         555
#define        GETARG         556
#define        PUTARG         557
#define        PROGEND        558
#define        ADDRARG        559
#define        START          560
#define        BRKPOINT       561
```

```
/*
--------------------------------------------------------------------------
!                                                                        !
!                    Alocacao do Modulo de Memoria                       !
!                                                                        !
--------------------------------------------------------------------------
*/

int code_memory[4096];              /* area destinada a estrutura de codigo */
int stack_memory[1024];             /* area destinada a estrutura de pilha  */
int variable_memory[1024];          /* area destinada a estrutura de variaveis */
int flag;                           /* flag indicativo de proc. "corner turning" */


/*
--------------------------------------------------------------------------
!                                                                        !
!                    * Rotinas genericas de apoio                        !
!                                                                        !
--------------------------------------------------------------------------
*/

void clr_screen(void)

{
 union REGS regs;
 regs.h.al=6;
 regs.h.ah=0;
 regs.h.ch=0;
 regs.h.cl=0;
 regs.h.dh=0;
 regs.h.dl=24;
 regs.h.bh=07;
 int86(0x10,&regs,&regs);         /*limpa a teal */
}


void print_bin(int valor,int tam)

{
  int i,j;

  if (tam==0) return;
  j=1<<(tam-1);
  for (i=0;i<tam;i++)
      {
      if (valor & j) printf("1");
                else printf("0");
      j=j>>1;
      }
}


/*
--------------------------------------------------------------------------
!                                                                        !
!     Modulo CARREGADOR:                                                 !
!                                                                        !
!             Executa a leitura do arquivo intermediario .LNK especificado, !
!     e recupera a partir do "header" informacoes para inicializacao das !
!     estruturas de memoria e dos registradores internos do gerador/inter- !
!     pretador.                                                          !
!             A passagem de parametros do modulo CARREGADOR para o gerador/ !
!     interpretador e' atraves da area de pilha. Este procedimento simula !
!     a passagem que ocorre no processador matricial proposto. As estrutu- !
!     ras de codigo, pilha e variavel compoem o módulo de memoria do pro- !
!     cessador proposto.                                                 !
!                                                                        !
--------------------------------------------------------------------------
*/
```

```
void carregador(void)

{
 struct ffblk diretorio;

 char pathname[18], *name;
 int chave,k,i,j,arq,header[11],*help,aux[2][7];
 void *buff;

/*----------------- entrada do arquivo intermediario -----------------*/

 clr_screen();
 printf("\n \n");
 printf("\n    +-----------------------------------------------------------------+");
 printf("\n    |                   Modulo CARREGADOR - VERSAO 1.00               |");
 printf("\n    |                                                      Abril 1989 |");
 printf("\n    +-----------------------------------------------------------------+");
 printf("\n \n");

 gotoxy(10,9);printf("Introduza o nome do arquivo intermediario a ser carregado:");
 do {
    gotoxy(15,11);printf(">Nome do Arquivo: ____________");
    gotoxy(32,11);
    pathname[0]=15;
    name=cgets(pathname);
    }
 while (findfirst(name,&diretorio,0)==-1);

/*-------------------- abre arquivo para leitura --------------------*/

 arq=open(name,(O_RDONLY|O_BINARY),S_IREAD);

/*----------------      leitura do "header"     ------------------------*/

 buff=(void*)calloc(11,sizeof(int));
 j=_read(arq,buff,22);
 help=(int*)buff;

 printf("\n \n *Header do arquivo:\n \n");
 for (i=0;i<11;i++)
     {
     header[i]=*help;
     printf("%5d",*help);
     help++;
     }
 free(buff);

 /*------------------ leitura da tabela de simbolos ------------------------*/

 printf("\n \n *Tabela de Simbolos: \n \n");
 buff=(void*)calloc(22,sizeof(char));
 chave=0;k=0;
 for (i=0;i<header[0];i++)
     {
     j=_read(arq,buff,22);
     name=(char*)buff;
     for (j=0;j<8;j++)
         {pathname[j]=*name;name++;}
     pathname[8]='\0';
     if (!(strcmp("corner_t",pathname))) {chave=1;k=0;};
     if (!(strcmp("_BUFFER",pathname))) {chave=1;k=1;};
     printf("%10s  ....   ",pathname);
     help=(int*)name;
     for (j=0;j<7;j++)
         {
         if (chave) aux[k][j]=*help;
```

```
        printf("%6d",*help);
        help++;
        }
    printf("\n");
    }
free(buff);


/*------- Leitura dos valores de inicializacao da memoria de variaveis ----*/

printf("\n * Inicializacao de Variaveis: \n \n");
printf("\nendereco  valor \n \n");
buff=(void*)calloc(header[2],sizeof(int));
help=(int*)buff;
j=_read(arq,buff,(header[2]*2));
j=0;
while (header[2])
      {
      printf("%6d  ",*help);
      variable_memory[j]=*help;
      j++;
      help++;
      header[2]--;
      }
free(buff);
printf("\n \n");


/*--------------------- Salta a tabela de labels ------------------------*/

buff=(void*)calloc(header[1],sizeof(int));
_read(arq,buff,(header[1]*2));
free(buff);


/*--------------------- Copia opcodes para code_memory ------------------*/


printf("\n \n *Opcodes em GAL: \n \n");
buff=(void*)calloc(header[3],sizeof(int));
j=_read(arq,buff,(header[3]*2));
help=(int*)buff;
printf("\n endereco    opcode    mnemonico GAL \n \n");

j=0;
while (header[3])
      {
      printf("\n %7.0d %10.0d     ",j,*help);
      code_memory[j]=*help;
      i=*help-500;
      if ((i)0)&&(i(62)) printf("%12s",op_code[i].opcode);
      help++;
      header[3]--;
      j++;
      }

printf("\n \n \n");



/*---------- Passagem de parametros via estrutura de pilha --------------*/


stack_memory[0]=aux[0][3];          /* inicio da rotina de corner turning */
stack_memory[1]=aux[1][4];          /* endereco inicial da variavel image */
stack_memory[2]=aux[1][5];          /* tamanho da variavel image */
stack_memory[3]=header[2];          /* inicializacao do auto_pointer */
stack_memory[4]=header[5];          /* inicializacao do add_pointer */
stack_memory[5]=header[10];         /* inicializacao do program_counter */
```

```
free(buff);
close(arq);
gotoxy(5,23);printf("\n ( FIM do procedimento de Carga - Pressione qualquer tecla para continuar )");
getch();

}

/*
------------------------------------------------------------------------------
Rotina de Interrupcao:

       Na versao assembly 8031 esta rotina deve estar associada a interrup-
cao do bit de E/S do registrador de comandos. Basicamente o que efetua e' o
salvamento completo do contexto do processador virtual, e a simulacao de u-
ma CALL. A rotina  associada  a  CALL e'  que efetua o "corner turning". O
RETURN da CALL simulada nao recebe o tratamento normal, no codigo abaixo es
ta distincao e' feita via a variavel "flag". O RETURN simulado restaura con
texto.
------------------------------------------------------------------------------

void interrupcao_E/S(void)

{
 1. Salvar contexto: pc, sp, arg_p, addr_p, gapp_inst
                     p_aux, op_aux, aux, ret_value

 2. Simular chamada da CALL:
       sp++;
       stack_memory[sp]=end_ram_gapp;
       sp++;
       stack_memory[sp]=tam_imagem;
       sp++;
       stack_memory[sp]=2 ; numero de argumentos
       sp++;
       stack_memory[sp]=0;  offset para addr_p;
       sp++;
       stack_memory[sp]=0;  offset para auto_p;
       sp++;
       stack_memory[sp]=end_corner_t;

       -------- executa a CALL ------------

       op_aux=pc;
       pc=stack_memory[sp];
       sp--;
       auto_p=auto_p+stack_memory[sp];
       sp--;
       addr_p=addr_p+stack_memory[sp];
       stack_memory[sp]=arg_p;
       sp++;
       stack_memory[sp]=0;
       sp++;
       stack_memory[sp]=0;
       sp==;
       stack_memory[sp]=op_aux;

3. setar flag de corner turning em execucao

}
*/



void main(void)

{

/*
------------------------------------------------------------------------------
```

```
!                                                                              !
!              Declaracao de variaveis do Interpretador                       !
!                                                                              !
--------------------------------------------------------------------------------
*/


/*------------------ Registradores Internos ---------------------------*/

int pc,                     /* program counter register */
    sp,                     /* stack pointer register */
    arg_p,                  /* argument pointer register */
    addr_p,                 /* address pointer register */
    auto_p;                /* auto pointer register */


/*------------ Registrador de Comando - Instrucao e Endereco -----------*/

int gapp_addr,              /* endereco da RAM interna do GAPP */
    gapp_inst;              /* comando para o GAPP - linhas de controle */


/*-- RAM interna do 8031 - Dados para simulacao da CALL de corner turning --*/

struct {
       int end_corner_t;
       int end_ram_gapp;
       int tam_imagem;
       } ram_interna;

/*------------------- Variaveis auxiliares ----------------------------*/

int p_aux,op_aux,aux,ret_value;    /* variaveis auxiliares */
int linha;
div_t x;



carregador();
clr_screen();
linha=0;


/*
--------------------------------------------------------------------------------
!                                                                              !
!                       Inicializacao do Interpretador                         !
!                                                                              !
--------------------------------------------------------------------------------
*/

auto_p=stack_memory[3];
addr_p=stack_memory[4];
pc    =stack_memory[5];
sp    =0;
arg_p =0;

gapp_addr=0;
gapp_inst=0;
flag=0;


/*--- Inicializacao da estrutura que simula a memoria interna do 8031 --*/

ram_interna.end_corner_t=stack_memory[0];
ram_interna.end_ram_gapp=stack_memory[1];
ram_interna.tam_imagem=stack_memory[2];
```

```
printf("\n+---------------------------------------------------------------------------+");
printf("\n|                                                                           |");
printf("\n|                   INTERPRETADOR/GERADOR - VERSAO 1.00                     |");
printf("\n|                                                                           |");
printf("\n|                                                        Maio 1989          |");
printf("\n+---------------------------------------------------------------------------+");
printf("\n \n \n");
printf("* Conteudo Registrador de Comandos: \n \n");
printf("              linha    Enderecos       Comandos \n");
printf("     Instrucao          6543210     CBA9876543210 \n");

while (code_memory[pc]!=PROGEND)
      {
      switch (code_memory[pc])
            {
            /*
            ---------------------------------------------------------------------------
            opcodes de operacao:
                   Opcodes responsaveis pela realizacao de operacoes aritmeticas, bole
            anas, e logicas. Os operandos sao valores do topo da pilha. O resultado  da
            operacao eh armazenado no topo da pilha.
            ---------------------------------------------------------------------------
            */

      case   OPBINMIN:      pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            stack_memory[sp]=stack_memory[sp]-op_aux;
                            break;

      case   OPBITAND:      pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            stack_memory[sp]=stack_memory[sp]&op_aux;
                            break;

      case   OPBITNOT:      pc++;
                            stack_memory[sp]=~stack_memory[sp];
                            break;

      case   OPBITOR:       pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            stack_memory[sp]=stack_memory[sp]|op_aux;
                            break;

      case   OPBITXOR:      pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            stack_memory[sp]=stack_memory[sp]^op_aux;
                            break;

      case   OPDIV:         pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            stack_memory[sp]=stack_memory[sp]/op_aux;
                            break;

      case   OPEQ:          pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            if (op_aux==stack_memory[sp]) stack_memory[sp]=1;
                                                     else stack_memory[sp]=0;
                            break;

      case   OPGT:          pc++;
                            op_aux=stack_memory[sp];
                            sp--;
                            if (stack_memory[sp]>op_aux) stack_memory[sp]=1;
                                                    else stack_memory[sp]=0;
                            break;
```

```
case  OPGTEQ:      pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if (stack_memory[sp]>=op_aux) stack_memory[sp]=1;
                                         else stack_memory[sp]=0;
                   break;

case  OPLOGAND:    pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if ((op_aux & stack_memory[sp])!=0) stack_memory[sp]=1;
                                              else stack_memory[sp]=0;
                   break;

case  OPLOGNOT:    pc++;
                   if (stack_memory[sp]!=0) stack_memory[sp]=1;
                                     else stack_memory[sp]=0;
                   break;

case  OPLOGOR:     pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if ((op_aux|stack_memory[sp])!=0) stack_memory[sp]=1;
                                            else stack_memory[sp]=0;
                   break;

case  OPLT:        pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if (stack_memory[sp]<op_aux) stack_memory[sp]=1;
                                        else stack_memory[sp]=0;
                   break;

case  OPLTEQ:      pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if (stack_memory[sp]<=op_aux) stack_memory[sp]=1;
                                         else stack_memory[sp]=0;
                   break;

case  OPMOD:       pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   x=div(stack_memory[sp],op_aux);
                   stack_memory[sp]=x.quot;
                   break;

case  OPMULT:      pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   stack_memory[sp]=stack_memory[sp]*op_aux;
                   break;

case  OPNOTEQ:     pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   if (op_aux==stack_memory[sp]) stack_memory[sp]=0;
                                         else stack_memory[sp]=1;
                   break;

case  OPPLUS:      pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   stack_memory[sp]=stack_memory[sp]+op_aux;
                   break;

case  OPSHL:       pc++;
                   op_aux=stack_memory[sp];
                   sp--;
                   stack_memory[sp]=stack_memory[sp]<<op_aux;
                   break;

case  OPSHR:       pc++;
```

```
                        op_aux=stack_memory[sp];
                        sp--;
                        stack_memory[sp]=stack_memory[sp]>>op_aux;
                        break;

case    OPUNMIN:        pc++;
                        stack_memory[sp]=-stack_memory[sp];
                        break;

        /*---------------------- fim de opcodes de operacao ----------------------*/


        /*
        ------------------------------------------------------------------------
        Opcodes de Salto:
                Opcodes empregados para controle de fluxo do programa, implementando
        saltos condicionais e incondicionais.
        ------------------------------------------------------------------------
        */

case    JMP:            pc=stack_memory[sp];
                        sp--;
                        break;

case    JMPEQ:          op_aux=stack_memory[sp];
                        sp--;
                        if (stack_memory[sp]) pc++;
                                    else pc=op_aux;
                        sp--;
                        break;

case    JMPGOHI:        op_aux=stack_memory[sp];
                        sp--;
                        /*
                        -------------------------------------------------------
                        A ser implementado apenas na versao assembly 8031, pois
                        depende do pino GO da matriz (pino T0,T1 do 8031).
                                if (T1==HIGH) pc=op_aux;
                                        else pc++;
                        -------------------------------------------------------
                        */
                        pc++; /*simplesmente continua*/
                        break;

case    JMPGOLOW:       op_aux=stack_memory[sp];
                        sp--;
                        /*
                        -------------------------------------------------------
                        A ser implementado apenas na versao assembly 8031, pois
                        depende do pino GO da matriz (pino T0,T1 do 8031).
                                if (T1==LOW) pc=op_aux;
                                        else pc++;
                        -------------------------------------------------------
                        */
                        pc++; /*simplesmente continua*/
                        break;

                        /*-------------------- fim dos opcodes de salto --------------------------*/


                        /*
                        ------------------------------------------------------------------------
                        Opcodes de controle da pilha:
                                Opcodes responsaveis pela manipulacao da pilha.
                        ------------------------------------------------------------------------
                        */

case    OPCONST:        pc++;
                        op_aux=code_memory[pc];
                        pc++;
                        sp++;
                        stack_memory[sp]=op_aux;
                        break;
```

```
case  REVSTK:       pc++;
                    op_aux=stack_memory[sp];
                    p_aux=sp;
                    p_aux--;
                    stack_memory[sp]=stack_memory[p_aux];
                    stack_memory[p_aux]=op_aux;
                    break;

case  POPSTK:       pc++;
                    sp--;
                    break;

      /*---------------- fim das operacoes de controle da pilha ----------------*/

      /*
      -----------------------------------------------------------------------
      Opcodes de "entry point" de rotinas:
              Opcodes empregados para implementar controle a chamada de rotinas.
      -----------------------------------------------------------------------
      */
case  CALL:         pc++;

                    op_aux=pc;
                    pc=stack_memory[sp];
                    sp--;
                    auto_p=auto_p+stack_memory[sp];
                    sp--;
                    addr_p=addr_p+stack_memory[sp];
                    stack_memory[sp]=arg_p;
                    sp++;
                    stack_memory[sp]=0;
                    sp++;
                    stack_memory[sp]=0;
                    sp++;
                    stack_memory[sp]=op_aux;
                    break;

case  ENTRY:        pc++;
                    arg_p=sp;
                    break;

case  START:        pc++;
                    break;

case  RESUME:       auto_p=auto_p-stack_memory[sp];
                    sp--;
                    addr_p=addr_p-stack_memory[sp];
                    sp--;
                    ret_value=stack_memory[sp];
                    sp--;
                    arg_p=stack_memory[sp];
                    sp--;
                    sp=sp-stack_memory[sp];
                    stack_memory[sp]=ret_value;
                    pc++;
                    break;

case  RETURN:       if (flag) {/*
                          Final Corner Turning: Simula retorno da CALL

                          1. restaurar contexto previamente salvo na interrupcao
                          2. desativar flag de corner turning
                          */
                          }
                    else  {
                          ret_value=stack_memory[sp];
                          sp--;
                          stack_memory[arg_p-2]=ret_value;
                          sp=arg_p;
                          pc=stack_memory[sp];
                          sp--;
```

```
                        sp--;
                        }
                break;

/*---------------- fim dos opcodes de "entry point" --------------------*/

/*
------------------------------------------------------------------------------
Opcode de E/S nas estruturas de memoria:
          Opcodes responsaveis pela leitura e escrita de valores nas estru-
turas de pilha e memoria.
------------------------------------------------------------------------------
*/

case  GETARG:   op_aux=stack_memory[sp];
                sp--;
                aux=stack_memory[arg_p-4];
                op_aux=stack_memory[arg_p-4-aux+op_aux];
                sp++;
                stack_memory[sp]=op_aux;
                pc++;
                break;

case  PUTARG:   op_aux=stack_memory[sp];
                sp--;
                aux=stack_memory[arg_p-4];
                op_aux=stack_memory[arg_p-4-aux+op_aux];
                stack_memory[op_aux]=stack_memory[sp];
                pc++;
                break;

case  GETAUTO:  op_aux=auto_p+stack_memory[sp];
                sp--;
                sp++;
                stack_memory[sp]=variable_memory[op_aux];
                pc++;
                break;

case  PUTAUTO:  op_aux=auto_p+stack_memory[sp];
                sp--;
                variable_memory[op_aux]=stack_memory[sp];
                pc++;
                break;

case  GETGLOB:  op_aux=stack_memory[sp];
                sp--;
                sp++;
                stack_memory[sp]=variable_memory[op_aux];
                pc++;
                break;

case  PUTGLOB:  op_aux=stack_memory[sp];
                sp--;
                variable_memory[op_aux]=stack_memory[sp];
                pc++;
                break;

/*------------------- fim dos opcodes de E/S em memoria ------------------*/

/*
------------------------------------------------------------------------------
Opcode de Interface com o GAPP:
          Opcodes empregados para gerar instrucoes para a matriz de processa-
dores.
------------------------------------------------------------------------------
*/

case  ADDRARG:  op_aux=stack_memory[sp];
                sp--;
                aux=stack_memory[arg_p-4];
```

UFRGS BIBLIOTECA

```
                        op_aux=stack_memory[arg_p-4-aux+op_aux];
                        sp++;
                        stack_memory[sp]=op_aux;
                        pc++;
                        break;

        case    ADDRAUTO:       op_aux=stack_memory[sp];
                        stack_memory[sp]=op_aux+addr_p;
                        pc++;
                        break;

        case    GAPPADDR:       gapp_addr=stack_memory[sp];
                        sp--;
                        pc++;
                        break;

        case    GAPPINST:       gapp_inst=stack_memory[sp];
                        sp--;
                        pc++;
                        linha++;
                        printf("\n %12.0d          ",linha);
                        print_bin(gapp_addr,7);
                        printf("       ");
                        print_bin(gapp_inst,13);
                        break;

                /*------------------ fim dos opcodes de interface com GAPP --------------*/

                /*
                ---------------------------------------------------------------------------
                Opcodes de Depuracao:
                        Opcodes empregados pelo interpretador/gerador na deteccao de qual
                porcao do programa esta sendo executada, visando depuracao.
                ---------------------------------------------------------------------------
                */

        case    BLOCKEND:       pc++;
                        pc++;
                        break;

        case    BLOCKSTART:     pc++;
                        pc++;
                        break;

        case    BRKPOINT:       pc++;
                        break;

        case    CONDEND:        pc++;
                        break;

        case    STATEND:        pc++;
                        break;


                default:        printf("\n )))) ERRO: opcode invalido!!!");
                        printf("\n \n    INTERPRETACAO CANCELADA");
                        exit(0);
                        break;
                }
        }

printf("\n \n > FIM DA INTERPRETACAO - Nenhum erro detectado \n");
}
```

# ANEXO 7

## Programas exemplos e simulação

Este anexo fornece o resultado obtida com o simulador do anexo 6. A saída representa o conjunto de comandos gerados para a matriz de processadores na execução de alguns programas exemplos. Estes programas exemplos foram retirados da biblioteca de funções GAL distribuída pela NCR Corporation em conjunto com o sistema de desenvolvimento GAPSYS (NCR45GDS1-MS).

```
/************************************************************ */
/*     ---------Copyright 1987 by NCR Corporation---------    */
/*                  Dayton, Ohio, USA                         */
/*                  All rights reserved.                      */
/*              Property of NCR Corporation.                  */
/************************************************************ */

/************************************************************
*   FILENAME:     addtc.gal
*                                                          *
*   ORIGIN:       Dave Morgan                              *
*                                                          *
*   MADEFROM:     /usr/acct/chrish/sccs/gallib/s.addtc.gal
*                 @(#)addtc.gal 1.1\t13:44:51 3/19/87
************************************************************ */

/*  Function:
*        addtc is a GAL program which adds two two's complement numbers.
*
*   Parameters:
*             image   A  =  1st operand
*             image   B  =  2nd operand
*             image   R  =  Result field
*
*   Additional Memory Used:
*
*   Execution Time:
*        4-bit images             15 cycles
*        8-bit images             27 cycles
*        16-bit images            51 cycles
*
*   Notes:
*        1) The result area, R, must be 1 bit larger than the larger of the
*           two operands. A general form of the declaration could be:
*
*           R: (size(A) > size(B) ? size(A)+1 : size(B)+1);
*
*/


addtc(A,B,R)

image A,B,R;

{
    int     i,                      /* standard counter */
            m = size(A),            /* size of 1st operand */
            n = size(B);            /* size of 2nd operand */

    A:0 ns:=ram c:=0;
    B:0 ew:=ram;
    R:0 ram:=sm c:=cy;

    for (i = 1; i < (m<n?m:n); i++)
    {
        A:i ns:=ram;
        B:i ew:=ram;
        R:i ram:=sm c:=cy;
    }
    for (i = (m<n?m:n); i < (m>n?m:n); i++)
    {
        if (m <= n) A:m-1 ns:=ram;
        else A:i ns:=ram;
        if (n <= m) B:n-1 ew:=ram;
        else B:i ew:=ram;
        R:i ram:=sm c:=cy;
    }
    A:m-1 ns:=ram;
    B:n-1 ew:=ram;
    R:(m>n?m:n) ram:=sm;
}
```

```
+--------------------------------------------------------------------------+
|                                                                          |
|             INTERPRETADOR/GERADOR - VERSAO 1.00                          |
|                                                                          |
|                                                        Maio 1989         |
+--------------------------------------------------------------------------+
```

* Conteudo Registrador de Comandos:

| linha Instrucao | Enderecos 6543210 | Comandos CBA9876543210 |
|---|---|---|
| 1 | 0000000 | 0011000000100 |
| 2 | 0001000 | 0000000100000 |
| 3 | 0010000 | 1110100000000 |
| 4 | 0000001 | 0000000000100 |
| 5 | 0001001 | 0000000100000 |
| 6 | 0010001 | 1110100000000 |
| 7 | 0000010 | 0000000000100 |
| 8 | 0001010 | 0000000100000 |
| 9 | 0010010 | 1110100000000 |
| 10 | 0000011 | 0000000000100 |
| 11 | 0001011 | 0000000100000 |
| 12 | 0010011 | 1110100000000 |
| 13 | 0000100 | 0000000000100 |
| 14 | 0001100 | 0000000100000 |
| 15 | 0010100 | 1110100000000 |
| 16 | 0000101 | 0000000000100 |
| 17 | 0001101 | 0000000100000 |
| 18 | 0010101 | 1110100000000 |
| 19 | 0000110 | 0000000000100 |
| 20 | 0001110 | 0000000100000 |
| 21 | 0010110 | 1110100000000 |
| 22 | 0000111 | 0000000000100 |
| 23 | 0001111 | 0000000100000 |
| 24 | 0010111 | 1110100000000 |
| 25 | 0000111 | 0000000000100 |
| 26 | 0001111 | 0000000100000 |
| 27 | 0011000 | 1100000000000 |

> FIM DA INTERPRETACAO - Nenhum erro detectado

C:\ADM>

```
/************************************************************* */
/*      ---------Copyright 1987 by NCR Corporation---------     */
/*                     Dayton, Ohio, USA                         */
/*                    All rights reserved.                       */
/*                 Property of NCR Corporation.                  */
/************************************************************* */
```


```
/*************************************************************
 *   FILENAME:     subtc.gal
 *                                                          *
 *   ORIGIN:       Dave Morgan                              *
 *                                                          *
 *   MADEFROM:     /usr/acct/chrish/sccs/gallib/s.subtc.gal
 *                 @(#)subtc.gal 1.1\t13:58:25 3/19/87
************************************************************* */

/*  Function:
 *        subtc is a GAL program which subtracts two two's complement numbers.
 *        The algorithm is:
 *
 *              A - B = R
 *
 *
 *   Parameters:
 *               image   A  =  1st operand
 *               image   B  =  2nd operand
 *               image   R  =  Result field
 *
 *   Additional Memory Used:
 *
 *   Execution Time:
 *        4-bit images              15 cycles
 *        8-bit images              27 cycles
 *        16-bit images             51 cycles
 *
 *   Notes:
 *
 *        1) R must one bit larger than the larger of A and B.
 */

subtc(A,B,R)

image A, B, R;
{
   int     i,                          /* standard counter */
           nA = size(A),               /* size of A operand */
           nB = size(B);               /* size of B operand */

   A:0 ns:=ram c:=0;                   /* subtract LSB w/o borrow */
   B:0 ew:=ram;
   R:0 ram:=sm c:=bw;

   if(nA >= nB)
       for(i = 1; i < nA; i++)
       {
           A:i ns:=ram;
           B:(i<nB ? i : nB-1) ew:=ram;  /* sign extend B for i>nB-1 */
           R:i ram:=sm c:=bw;
       }
   else
       for(i = 1; i < nB; i++)
       {
           A:(i<nA ? i : nA-1) ns:=ram;  /* sign extend A for i>nA-1 */
           B:i ew:=ram;
           R:i ram:=sm c:=bw;
       }

   A:nA-1 ns:=ram;                    /* sign extend R to allow for overflow */
   B:nB-1 ew:=ram;
   R:(nA>nB ? nA : nB) ram:=sm;
}
```

```
+------------------------------------------------------------------+
|                                                                  |
|            INTERPRETADOR/GERADOR - VERSAO 1.00                   |
|                                                                  |
|                                              Maio 1989           |
+------------------------------------------------------------------+
```

* Conteudo Registrador de Comandos:

| linha Instrucao | Enderecos 6543210 | Comandos CBA9876543210 |
|---|---|---|
| 1 | 0000000 | 0011000000100 |
| 2 | 0001000 | 0000000100000 |
| 3 | 0010000 | 1110000000000 |
| 4 | 0000001 | 0000000000100 |
| 5 | 0001001 | 0000000100000 |
| 6 | 0010001 | 1110000000000 |
| 7 | 0000010 | 0000000000100 |
| 8 | 0001010 | 0000000100000 |
| 9 | 0010010 | 1110000000000 |
| 10 | 0000011 | 0000000000100 |
| 11 | 0001011 | 0000000100000 |
| 12 | 0010011 | 1110000000000 |
| 13 | 0000100 | 0000000000100 |
| 14 | 0001100 | 0000000100000 |
| 15 | 0010100 | 1110000000000 |
| 16 | 0000101 | 0000000000100 |
| 17 | 0001101 | 0000000100000 |
| 18 | 0010101 | 1110000000000 |
| 19 | 0000110 | 0000000000100 |
| 20 | 0001110 | 0000000100000 |
| 21 | 0010110 | 1110000000000 |
| 22 | 0000111 | 0000000000100 |
| 23 | 0001111 | 0000000100000 |
| 24 | 0010111 | 1110000000000 |
| 25 | 0000111 | 0000000000100 |
| 26 | 0001111 | 0000000100000 |
| 27 | 0011000 | 1100000000000 |

) FIM DA INTERPRETACAO - Nenhum erro detectado

C:\ADM)

```
/************************************************************ */
/*        ----------Copyright 1987 by NCR Corporation---------        */
/*                       Dayton, Ohio, USA                            */
/*                       All rights reserved.                         */
/*                    Property of NCR Corporation.                    */
/************************************************************ */

/**********************************************************
*   FILENAME:      dividesm.gal
*                                                              *
*   ORIGIN:        Dave Morgan                                 *
*                                                              *
*   MADEFROM:      /usr/accl/chrish/sccs/gallib/s.dividesm.gal
*                  @(#)dividesm.gal      1.1\t13:46:37 3/19/87
********************************************************** */

/*  Function:
*        dividesm is a GAL program which divides two signed magnitude numbers.
*
*   Parameters:
*                 image  P  =  Dividend
*                 image  Q  =  Divisor
*                 image  R  =  Result field
*                 image  S  =  Scratch area
*                 image  X  =  Single bit scratch area
*
*   Additional Memory Used:
*
*        This routine uses one (1) additional GAPP RAM location.
*
*   Execution Time:
*        4-bit images              84 cycles
*        8-bit images              402 cycles
*        16-bit images             1758 cycles
*
*   Notes:
*
*        1) The size of the result field, R, is size(P).
*        2) On exit, R contains the result (an interger) and P contains
*           the remainder.
*        3) R must be cleared (contain zero) before this routine is called.
*        4) S is the same size as the larger of P and Q.
*/

dividesm(Q, P, R, S)

image   P,                              /* dividend */
        Q,                              /* divisor */
        R,                              /* result area */
        S;                              /* scratch area */

{
    image   X:1;                /* single scratch bit */

    int     i,j;                /* standard counters */

    int     p = size(P);        /* size of dividend */
    int     q = size(Q);        /* size of divisor */

    for(j = p-2; j >= 0; j--)
    {
        Q:0 c:=0 ew:=ram;
        P:j ns:=ram;
        S:0 ram:=sm c:=bw;

        for (i = 1; i < q-1; i++)
        {
            Q:i ew:=ram;
            if (j+i > p-2) ns:=0;
            else P:j+i ns:=ram;
            S:i ram:=sm;
```

```
        c:=bw;
    }

    /* invert and store borrow */

    X: ram:=c ns:=0 ew:=c c:=1;
    R:j ram:=sm ns:=ew;

    for(i = 0; i+j < p-1; i++)
    {
        S:i ew:=ram c:=0;
        c:=bw;
        S:i ram:=c;
        P:j+i ew:=ram c:=0;
        S:i c:=cy ew:=ram ns:=0;
        X: c:=bw ns:=ram;
        P:j+i ram:=c;
    }
  }

  /* calculate sign */

  Q:q-1 ns:=ram;
  P:p-1 ew:=ram c:=0;
  R:p-1 ram:=sm;
}
```

```
+------------------------------------------------------------------+
|                                                                  |
|          INTERPRETADOR/GERADOR - VERSAO 1.00                     |
|                                                                  |
|                                             Maio 1989           |
+------------------------------------------------------------------+
```

* Conteudo Registrador de Comandos:

| Linha Instrucao | Enderecos 6543210 | Comandos CBA9876543210 |
|---|---|---|
| 1 | 0000000 | 0011000100000 |
| 2 | 0001110 | 0000000000100 |
| 3 | 0011000 | 1110100000000 |
| 4 | 0000001 | 0000000100000 |
| 5 | 0000001 | 0000000011000 |
| 6 | 0011001 | 1100000000000 |
| 7 | 0011001 | 0010100000000 |
| 8 | 0000010 | 0000000100000 |
| 9 | 0000010 | 0000000011000 |
| 10 | 0011010 | 1100000000000 |
| 11 | 0011010 | 0010100000000 |
| 12 | 0000011 | 0000000100000 |
| 13 | 0000011 | 0000000011000 |
| 14 | 0011011 | 1100000000000 |
| 15 | 0011011 | 0010100000000 |
| 16 | 0000100 | 0000000100000 |
| 17 | 0000100 | 0000000011000 |
| 18 | 0011100 | 1100000000000 |
| 19 | 0011100 | 0010100000000 |
| 20 | 0000101 | 0000000100000 |
| 21 | 0000101 | 0000000011000 |
| 22 | 0011101 | 1100000000000 |
| 23 | 0011101 | 0010100000000 |
| 24 | 0000110 | 0000000100000 |
| 25 | 0000110 | 0000000011000 |
| 26 | 0011110 | 1100000000000 |
| 27 | 0011110 | 0010100000000 |
| 28 | 0011010 | 1011110111000 |
| 29 | 0010110 | 1100000010000 |
| 30 | 0011000 | 0011000100000 |
| 31 | 0011000 | 0010100000000 |
| 32 | 0011000 | 1000000000000 |
| 33 | 0001110 | 0011000100000 |
| 34 | 0011000 | 0010000111000 |
| 35 | 0011010 | 0010100000100 |
| 36 | 0001110 | 1000000000000 |
| 37 | 0000000 | 0011000100000 |
| 38 | 0001101 | 0000000000100 |
| 39 | 0011000 | 1110100000000 |
| 40 | 0000001 | 0000000100000 |
| 41 | 0001110 | 0000000000100 |
| 42 | 0011001 | 1100000000000 |
| 43 | 0011001 | 0010100000000 |
| 44 | 0000010 | 0000000100000 |
| 45 | 0000010 | 0000000011000 |
| 46 | 0011010 | 1100000000000 |
| 47 | 0011010 | 0010100000000 |
| 48 | 0000011 | 0000000100000 |
| 49 | 0000011 | 0000000011000 |
| 50 | 0011011 | 1100000000000 |
| 51 | 0011011 | 0010100000000 |
| 52 | 0000100 | 0000000100000 |
| 53 | 0000100 | 0000000011000 |
| 54 | 0011100 | 1100000000000 |
| 55 | 0011100 | 001010000000 |
| 56 | 0000101 | 000000010000 |
| 57 | 0000101 | 000000001100 |
| 58 | 0011101 | 110000000000 |
| 59 | 0011101 | 001010000000 |
| 60 | 0000110 | 000000010000 |
| 61 | 0000110 | 000000001100 |
| 62 | 0011110 | 110000000000 |
| 63 | 0011110 | 001010000000 |
| 64 | 0011010 | 101111011100 |
| 65 | 0010101 | 110000001000 |
| 66 | 0011000 | 001100010000 |
| 67 | 0011000 | 001010000000 |
| 68 | 0011000 | 100000000000 |
| 69 | 0001101 | 001100010000 |
| 70 | 0011000 | 001000011100 |
| 71 | 0011010 | 001010000010 |
| 72 | 0001101 | 100000000000 |
| 73 | 0011001 | 001100010000 |
| 74 | 0011001 | 001010000000 |
| 75 | 0011001 | 100000000000 |
| 76 | 0001110 | 001100010000 |
| 77 | 0011001 | 001000011100 |
| 78 | 0011010 | 001010000010 |
| 79 | 0001110 | 100000000000 |
| 80 | 0000000 | 001100010000 |
| 81 | 0001100 | 000000000010 |
| 82 | 0011000 | 111010000000 |
| 83 | 0000001 | 000000010000 |
| 84 | 0001101 | 000000000010 |
| 85 | 0011001 | 110000000000 |
| 86 | 0011001 | 001010000000 |
| 87 | 0000010 | 000000010000 |
| 88 | 0001110 | 000000000010 |
| 89 | 0011010 | 110000000000 |
| 90 | 0011010 | 001010000000 |
| 91 | 0000011 | 000000010000 |
| 92 | 0000011 | 000000001100 |
| 93 | 0011011 | 110000000000 |
| 94 | 0011011 | 001010000000 |
| 95 | 0000100 | 000000010000 |
| 96 | 0000100 | 000000001100 |
| 97 | 0011100 | 110000000000 |
| 98 | 0011100 | 001010000000 |
| 99 | 0000101 | 000000010000 |
| 100 | 0000101 | 000000001100 |
| 101 | 0011101 | 110000000000 |
| 102 | 0011101 | 001010000000 |
| 103 | 0000110 | 000000010000 |
| 104 | 0000110 | 000000001100 |
| 105 | 0011110 | 110000000000 |
| 106 | 0011110 | 001010000000 |
| 107 | 0011010 | 101111011100 |
| 108 | 0010100 | 110000001000 |

| | | |
|---|---|---|
| 109 | 0011000 | 0011000100000 |
| 110 | 0011000 | 0010100000000 |
| 111 | 0011000 | 1000000000000 |
| 112 | 0001100 | 0011000100000 |
| 113 | 0011000 | 0010000111000 |
| 114 | 0011010 | 0010100000100 |
| 115 | 0001100 | 1000000000000 |
| 116 | 0011001 | 0011000100000 |
| 117 | 0011001 | 0010100000000 |
| 118 | 0011001 | 1000000000000 |
| 119 | 0001101 | 0011000100000 |
| 120 | 0011001 | 0010000111000 |
| 121 | 0011010 | 0010100000100 |
| 122 | 0001101 | 1000000000000 |
| 123 | 0011010 | 0011000100000 |
| 124 | 0011010 | 0010100000000 |
| 125 | 0011010 | 1000000000000 |
| 126 | 0001110 | 0011000100000 |
| 127 | 0011010 | 0010000111000 |
| 128 | 0011010 | 0010100000100 |
| 129 | 0001110 | 1000000000000 |
| 130 | 0000000 | 0011000100000 |
| 131 | 0001011 | 0000000000100 |
| 132 | 0011000 | 1110100000000 |
| 133 | 0000001 | 0000000100000 |
| 134 | 0001100 | 0000000000100 |
| 135 | 0011001 | 1100000000000 |
| 136 | 0011001 | 0010100000000 |
| 137 | 0000010 | 0000000100000 |
| 138 | 0001101 | 0000000000100 |
| 139 | 0011010 | 1100000000000 |
| 140 | 0011010 | 0010100000000 |
| 141 | 0000011 | 0000000100000 |
| 142 | 0001110 | 0000000000100 |
| 143 | 0011011 | 1100000000000 |
| 144 | 0011011 | 0010100000000 |
| 145 | 0000100 | 0000000100000 |
| 146 | 0000100 | 0000000011000 |
| 147 | 0011100 | 1100000000000 |
| 148 | 0011100 | 0010100000000 |
| 149 | 0000101 | 0000000100000 |
| 150 | 0000101 | 0000000011000 |
| 151 | 0011101 | 1100000000000 |
| 152 | 0011101 | 0010100000000 |
| 153 | 0000110 | 0000000100000 |
| 154 | 0000110 | 0000000011000 |
| 155 | 0011110 | 1100000000000 |
| 156 | 0011110 | 0010100000000 |
| 157 | 0011010 | 1011110111000 |
| 158 | 0010011 | 1100000010000 |
| 159 | 0011000 | 0011000100000 |
| 160 | 0011000 | 0010100000000 |
| 161 | 0011000 | 1000000000000 |
| 162 | 0001011 | 0011000100000 |
| 163 | 0011000 | 0010000111000 |
| 164 | 0011010 | 0010100000100 |
| 165 | 0001011 | 1000000000000 |
| 166 | 0011001 | 0011000100000 |
| 167 | 0011001 | 0010100000000 |
| 168 | 0011001 | 1000000000000 |
| 169 | 0001100 | 0011000100000 |
| 170 | 0011001 | 0010000111000 |
| 171 | 0011010 | 0010100000100 |
| 172 | 0001100 | 1000000000000 |
| 173 | 0011010 | 0011000100000 |
| 174 | 0011010 | 0010100000000 |
| 175 | 0011010 | 1000000000000 |
| 176 | 0001101 | 0011000100000 |
| 177 | 0011010 | 0010000111000 |
| 178 | 0011010 | 0010100000100 |
| 179 | 0001101 | 1000000000000 |
| 180 | 0011011 | 0011000100000 |
| 181 | 0011011 | 0010100000000 |
| 182 | 0011011 | 1000000000000 |
| 183 | 0001110 | 0011000100000 |
| 184 | 0011011 | 0010000111000 |
| 185 | 0011010 | 0010100000100 |
| 186 | 0001110 | 1000000000000 |
| 187 | 0000000 | 0011000100000 |
| 188 | 0001010 | 0000000000100 |
| 189 | 0011000 | 1110100000000 |
| 190 | 0000001 | 0000000100000 |
| 191 | 0001011 | 0000000000100 |
| 192 | 0011001 | 1100000000000 |
| 193 | 0011001 | 0010100000000 |
| 194 | 0000010 | 0000000100000 |
| 195 | 0001100 | 0000000000100 |
| 196 | 0011010 | 1100000000000 |
| 197 | 0011010 | 0010100000000 |
| 198 | 0000011 | 0000000100000 |
| 199 | 0001101 | 0000000000100 |
| 200 | 0011011 | 1100000000000 |
| 201 | 0011011 | 0010100000000 |
| 202 | 0000100 | 0000000100000 |
| 203 | 0001110 | 0000000000100 |
| 204 | 0011100 | 1100000000000 |
| 205 | 0011100 | 0010100000000 |
| 206 | 0000101 | 0000000100000 |
| 207 | 0000101 | 0000000011000 |
| 208 | 0011101 | 1100000000000 |
| 209 | 0011101 | 0010100000000 |
| 210 | 0000110 | 0000000100000 |
| 211 | 0000110 | 0000000011000 |
| 212 | 0011110 | 1100000000000 |
| 213 | 0011110 | 0010100000000 |
| 214 | 0011010 | 1011110111000 |
| 215 | 0010010 | 1100000010000 |
| 216 | 0011000 | 0011000100000 |
| 217 | 0011000 | 0010100000000 |
| 218 | 0011000 | 1000000000000 |
| 219 | 0001010 | 0011000100000 |
| 220 | 0011000 | 0010000111000 |
| 221 | 0011010 | 0010100000100 |
| 222 | 0001010 | 1000000000000 |
| 223 | 0011001 | 0011000100000 |
| 224 | 0011001 | 0010100000000 |
| 225 | 0011001 | 1000000000000 |
| 226 | 0001011 | 0011000100000 |
| 227 | 0011001 | 0010000111000 |
| 228 | 0011010 | 0010100000100 |
| 229 | 0001011 | 1000000000000 |
| 230 | 0011010 | 0011000100000 |
| 231 | 0011010 | 0010100000000 |
| 232 | 0011010 | 1000000000000 |
| 233 | 0001100 | 0011000100000 |
| 234 | 0011010 | 0010000111000 |
| 235 | 0011010 | 0010100000100 |
| 236 | 0001100 | 1000000000000 |
| 237 | 0011011 | 0011000100000 |
| 238 | 0011011 | 0010100000000 |
| 239 | 0011011 | 1000000000000 |
| 240 | 0001101 | 0011000100000 |
| 241 | 0011011 | 0010000111000 |
| 242 | 0011010 | 0010100000100 |
| 243 | 0001101 | 1000000000000 |
| 244 | 0011100 | 0011000100000 |
| 245 | 0011100 | 0010100000000 |
| 246 | 0011100 | 1000000000000 |
| 247 | 0001110 | 0011000100000 |
| 248 | 0011100 | 0010000111000 |
| 249 | 0011010 | 0010100000100 |
| 250 | 0001110 | 1000000000000 |
| 251 | 0000000 | 0011000100000 |
| 252 | 0001001 | 0000000000100 |
| 253 | 0011000 | 1110100000000 |
| 254 | 0000001 | 0000000100000 |
| 255 | 0001010 | 0000000000100 |
| 256 | 0011001 | 1100000000000 |
| 257 | 0011001 | 0010100000000 |
| 258 | 0000010 | 0000000100000 |

```
259    0001011    0000000000100
260    0011010    1100000000000
261    0011010    0010100000000
262    0000011    0000000100000
263    0001100    0000000000100
264    0011011    1100000000000
265    0011011    0010100000000
266    0000100    0000000100000
267    0001101    0000000000100
268    0011100    1100000000000
269    0011100    0010100000000
270    0000101    0000000100000
271    0001110    0000000000100
272    0011101    1100000000000
273    0011101    0010100000000
274    0000110    0000000100000
275    0000110    0000000011000
276    0011110    1100000000000
277    0011110    0010100000000
278    0011010    1011110111000
279    0010001    1100000010000
280    0011000    0011000100000
281    0011000    0010100000000
282    0011000    1000000000000
283    0001001    0011000100000
284    0011000    0010000111000
285    0011010    0010100000100
286    0001001    1000000000000
287    0011001    0011000100000
288    0011001    0010100000000
289    0011001    1000000000000
290    0001010    0011000100000
291    0011001    0010000111000
292    0011010    0010100000100
293    0001010    1000000000000
294    0011010    0011000100000
295    0011010    0010100000000
296    0011010    1000000000000
297    0001011    0011000100000
298    0011010    0010000111000
299    0011010    0010100000100
300    0001011    1000000000000
301    0011011    0011000100000
302    0011011    0010100000000
303    0011011    1000000000000
304    0001100    0011000100000
305    0011011    0010000111000
306    0011010    0010100000100
307    0001100    1000000000000
308    0011100    0011000100000
309    0011100    0010100000000
310    0011100    1000000000000
311    0001101    0011000100000
312    0011100    0010000111000
313    0011010    0010100000100
314    0001101    1000000000000
315    0011101    0011000100000
316    0011101    0010100000000
317    0011101    1000000000000
318    0001110    0011000100000
319    0011101    0010000111000
320    0011010    0010100000100
321    0001110    1000000000000
322    0000000    0011000100000
323    0001000    0000000000100
324    0011000    1110100000000
325    0000001    0000000100000
326    0001001    0000000000100
327    0011001    1100000000000
328    0011001    0010100000000
329    0000010    0000000100000
330    0001010    0000000000100
331    0011010    1100000000000
332    0011010    0010100000000
333    0000011    0000000100000
334    0001011    0000000000100
335    0011011    1100000000000
336    0011011    0010100000000
337    0000100    0000000100000
338    0001100    0000000000100
339    0011100    1100000000000
340    0011100    0010100000000
341    0000101    0000000100000
342    0001101    0000000000100
343    0011101    1100000000000
344    0011101    0010100000000
345    0000110    0000000100000
346    0001110    0000000000100
347    0011110    1100000000000
348    0011110    0010100000000
349    0011010    1011110111000
350    0010000    1100000010000
351    0011000    0011000100000
352    0011000    0010100000000
353    0011000    1000000000000
354    0001000    0011000100000
355    0011000    0010000111000
356    0011010    0010100000100
357    0001000    1000000000000
358    0011001    0011000100000
359    0011001    0010100000000
360    0011001    1000000000000
361    0001001    0011000100000
362    0011001    0010000111000
363    0011010    0010100000100
364    0001001    1000000000000
365    0011010    0011000100000
366    0011010    0010100000000
367    0011010    1000000000000
368    0001010    0011000100000
369    0011010    0010000111000
370    0011010    0010100000100
371    0001010    1000000000000
372    0011011    0011000100000
373    0011011    0010100000000
374    0011011    1000000000000
375    0001011    0011000100000
376    0011011    0010000111000
377    0011010    0010100000100
378    0001011    1000000000000
379    0011100    0011000100000
380    0011100    0010100000000
381    0011100    1000000000000
382    0001100    0011000100000
383    0011100    0010000111000
384    0011010    0010100000100
385    0001100    1000000000000
386    0011101    0011000100000
387    0011101    0010100000000
388    0011101    1000000000000
389    0001101    0011000100000
390    0011101    0010000111000
391    0011010    0010100000100
392    0001101    1000000000000
393    0011110    0011000100000
394    0011110    0010100000000
395    0011110    1000000000000
396    0001110    0011000100000
397    0011110    0010000111000
398    0011010    0010100000100
399    0001110    1000000000000
400    0000111    0000000000100
401    0001111    0011000100000
402    0010111    1100000000000
```

) FIM DA INTERPRETACAO - Nenhum erro detectado

```
/*********************************************************** */
/*      ---------Copyright 1987 by NCR Corporation---------     */
/*                   Dayton, Ohio, USA                           */
/*                  All rights reserved.                         */
/*               Property of NCR Corporation.                    */
/*********************************************************** */
```


```
/**********************************************************
*   FILENAME:     multsm.gal
*                                                          *
*   ORIGIN:       Dave Morgan                              *
*                                                          *
*   MADEFROM:     /usr/acct/chrish/sccs/gallib/s.multsm.gal
*                 @(#)multsm.gal       1.1\t13:52:09 3/19/87
*********************************************************** */

/*  Function:
*        multsm is a GAL program which multiplies two signed magnitude numbers.
*
*   Parameters:
*               image  A  =  Multiplicand
*               image  B  =  Multiplier
*               image  C  =  Result field
*               image  X  =  Scratch area
*
*   Additional Memory Used:
*
*        This routine uses one (1) additional GAPP RAM location.
*
*   Execution Time:
*        4-bit images              47 cycles
*        8-bit images              271 cycles
*        16-bit images             1275 cycles
*
*   Notes:
*
*        1) The size of the result field is at least m + n - 1,
*           where n and m are the sizes of A and B respectively.
*/

multsm(A,B,C)

image   A,                     /* multiplicand */
        B,                     /* multiplier */
        C;                     /* result area */
{
    image   X:1;                   /* one bit scratch area */
    int     h,i,                   /* standard counters */
            n = size A,            /* size of multiplicand */
            m = size B;            /* size of multplier */

    B:0 ew:=ram;

    for (i = 0; i < n-1; i++)
    {
        A:i ns:=ram c:=0;
        c:=cy;
        C:i ram:=c;
    }

    for (h = 1; h < m-1; h++)
    {
        A:0 ns:=ram c:=0;
        B:h ew:=ram;
        C:h ns:=ram ew:=0 c:=cy;
        C:h ram:=sm c:=cy;
        X: ram:=c;

        for (i = 1; i < n-2; i++)
        {
            A:i ns:=ram c:=0;
```

```
            B:h ew:=ram;
            X: ew:=ram c:=cy;
            C:i+h ns:=ram;
            C:i+h ram:=sm c:=cy;
            X: ram:=c;
        }

        A:n-2 ns:=ram c:=0;
        B:h ew:=ram;
        X: ew:=ram c:=cy;
        C:n+h-2 ns:=ram;
        C:n+h-2 ram:=sm c:=cy;
        C:n+h-1 ram:=c;
    }
    A:n-1 ns:=ram;
    B:m-1 ew:=ram c:=0;
    C:n+m-2 ram:=sm;
}
```

UFROS
BIBLIOTECA
CPD

```
+--------------------------------------------------------------+
|                                                              |
|         INTERPRETADOR/GERADOR - VERSAO 1.00                  |
|                                                              |
|                                          Maio 1989           |
+--------------------------------------------------------------+
```

* Conteudo Registrador de Comandos:

| Linha Instrucao | Enderecos 6543210 | Comandos CBA9876543210 |
|---|---|---|
| 1 | 0001000 | 0000000100000 |
| 2 | 0000000 | 0011000000100 |
| 3 | 0000000 | 0010000000000 |
| 4 | 0010000 | 1000000000000 |
| 5 | 0000001 | 0011000000100 |
| 6 | 0000001 | 0010000000000 |
| 7 | 0010001 | 1000000000000 |
| 8 | 0000010 | 0011000000100 |
| 9 | 0000010 | 0010000000000 |
| 10 | 0010010 | 1000000000000 |
| 11 | 0000011 | 0011000000100 |
| 12 | 0000011 | 0010000000000 |
| 13 | 0010011 | 1000000000000 |
| 14 | 0000100 | 0011000000100 |
| 15 | 0000100 | 0010000000000 |
| 16 | 0010100 | 1000000000000 |
| 17 | 0000101 | 0011000000100 |
| 18 | 0000101 | 0010000000000 |
| 19 | 0010101 | 1000000000000 |
| 20 | 0000110 | 0011000000100 |
| 21 | 0000110 | 0010000000000 |
| 22 | 0010110 | 1000000000000 |
| 23 | 0000000 | 0011000000100 |
| 24 | 0001001 | 0000000100000 |
| 25 | 0010001 | 0010011000100 |
| 26 | 0010001 | 1110000000000 |
| 27 | 0100000 | 1000000000000 |
| 28 | 0000001 | 0011000000100 |
| 29 | 0001001 | 0000000100000 |
| 30 | 0100000 | 0010000100000 |
| 31 | 0010010 | 0000000000100 |
| 32 | 0010010 | 1110000000000 |
| 33 | 0100000 | 1000000000000 |
| 34 | 0000010 | 0011000000100 |
| 35 | 0001001 | 0000000100000 |
| 36 | 0100000 | 0010000100000 |
| 37 | 0010011 | 0000000000100 |
| 38 | 0010011 | 1110000000000 |
| 39 | 0100000 | 1000000000000 |
| 40 | 0000011 | 0011000000100 |
| 41 | 0001001 | 0000000100000 |
| 42 | 0100000 | 0010000100000 |
| 43 | 0010100 | 0000000000100 |
| 44 | 0010100 | 1110000000000 |
| 45 | 0100000 | 1000000000000 |
| 46 | 0000100 | 0011000000100 |
| 47 | 0001001 | 0000000100000 |
| 48 | 0100000 | 0010000100000 |
| 49 | 0010101 | 0000000000100 |
| 50 | 0010101 | 1110000000000 |
| 51 | 0100000 | 1000000000000 |
| 52 | 0000101 | 0011000000100 |
| 53 | 0001001 | 0000000100000 |
| 54 | 0100000 | 0010000100000 |
| 55 | 0010110 | 0000000000100 |
| 56 | 0010110 | 1110000000000 |
| 57 | 0100000 | 1000000000000 |
| 58 | 0000110 | 0011000000100 |
| 59 | 0001001 | 0000000100000 |
| 60 | 0100000 | 0010000100000 |
| 61 | 0010111 | 0000000000100 |
| 62 | 0010111 | 1110000000000 |
| 63 | 0011000 | 1000000000000 |
| 64 | 0000000 | 0011000000100 |
| 65 | 0001010 | 0000000100000 |
| 66 | 0010010 | 0010011000100 |
| 67 | 0010010 | 1110000000000 |
| 68 | 0100000 | 1000000000000 |
| 69 | 0000001 | 0011000000100 |
| 70 | 0001010 | 0000000100000 |
| 71 | 0100000 | 0010000100000 |
| 72 | 0010011 | 0000000000100 |
| 73 | 0010011 | 1110000000000 |
| 74 | 0100000 | 1000000000000 |
| 75 | 0000010 | 0011000000100 |
| 76 | 0001010 | 0000000100000 |
| 77 | 0100000 | 0010000100000 |
| 78 | 0010100 | 0000000000100 |
| 79 | 0010100 | 1110000000000 |
| 80 | 0100000 | 1000000000000 |
| 81 | 0000011 | 0011000000100 |
| 82 | 0001010 | 0000000100000 |
| 83 | 0100000 | 0010000100000 |
| 84 | 0010101 | 0000000000100 |
| 85 | 0010101 | 1110000000000 |
| 86 | 0100000 | 1000000000000 |
| 87 | 0000100 | 0011000000100 |
| 88 | 0001010 | 0000000100000 |
| 89 | 0100000 | 0010000100000 |
| 90 | 0010110 | 0000000000100 |
| 91 | 0010110 | 1110000000000 |
| 92 | 0100000 | 1000000000000 |
| 93 | 0000101 | 0011000000100 |
| 94 | 0001010 | 0000000100000 |
| 95 | 0100000 | 0010000100000 |
| 96 | 0010111 | 0000000000100 |
| 97 | 0010111 | 1110000000000 |
| 98 | 0100000 | 1000000000000 |
| 99 | 0000110 | 0011000000100 |
| 100 | 0001010 | 0000000100000 |
| 101 | 0100000 | 0010000100000 |
| 102 | 0011000 | 0000000000100 |
| 103 | 0011000 | 1110000000000 |
| 104 | 0011001 | 1000000000000 |
| 105 | 0000000 | 0011000000100 |
| 106 | 0001011 | 0000000100000 |
| 107 | 0010011 | 0010011000100 |
| 108 | 0010011 | 1110000000000 |
| 109 | 0100000 | 1000000000000 |
| 110 | 0000001 | 0011000000100 |
| 111 | 0001011 | 0000000100000 |
| 112 | 0100000 | 0010000100000 |
| 113 | 0010100 | 0000000000100 |
| 114 | 0010100 | 1110000000000 |
| 115 | 0100000 | 1000000000000 |
| 116 | 0000010 | 0011000000100 |
| 117 | 0001011 | 0000000100000 |
| 118 | 0100000 | 0010000100000 |
| 119 | 0010101 | 0000000000100 |
| 120 | 0010101 | 1110000000000 |
| 121 | 0100000 | 1000000000000 |
| 122 | 0000011 | 0011000000100 |
| 123 | 0001011 | 0000000100000 |
| 124 | 0100000 | 0010000100000 |

| | | |
|---|---|---|
| 125 | 0010110 | 0000000000100 |
| 126 | 0010110 | 1110000000000 |
| 127 | 0100000 | 1000000000000 |
| 128 | 0000100 | 0011000000100 |
| 129 | 0001011 | 0000000100000 |
| 130 | 0100000 | 0010000100000 |
| 131 | 0010111 | 0000000000100 |
| 132 | 0010111 | 1110000000000 |
| 133 | 0100000 | 1000000000000 |
| 134 | 0000101 | 0011000000100 |
| 135 | 0001011 | 0000000100000 |
| 136 | 0100000 | 0010000100000 |
| 137 | 0011000 | 0000000000100 |
| 138 | 0011000 | 1110000000000 |
| 139 | 0100000 | 1000000000000 |
| 140 | 0000110 | 0011000000100 |
| 141 | 0001011 | 0000000100000 |
| 142 | 0100000 | 0010000100000 |
| 143 | 0011001 | 0000000000100 |
| 144 | 0011001 | 1110000000000 |
| 145 | 0011010 | 1000000000000 |
| 146 | 0000000 | 0011000000100 |
| 147 | 0001100 | 0000000100000 |
| 148 | 0010100 | 0010011000100 |
| 149 | 0010100 | 1110000000000 |
| 150 | 0100000 | 1000000000000 |
| 151 | 0000001 | 0011000000100 |
| 152 | 0001100 | 0000000100000 |
| 153 | 0100000 | 0010000100000 |
| 154 | 0010101 | 0000000000100 |
| 155 | 0010101 | 1110000000000 |
| 156 | 0100000 | 1000000000000 |
| 157 | 0000010 | 0011000000100 |
| 158 | 0001100 | 0000000100000 |
| 159 | 0100000 | 0010000100000 |
| 160 | 0010110 | 0000000000100 |
| 161 | 0010110 | 1110000000000 |
| 162 | 0100000 | 1000000000000 |
| 163 | 0000011 | 0011000000100 |
| 164 | 0001100 | 0000000100000 |
| 165 | 0100000 | 0010000100000 |
| 166 | 0010111 | 0000000000100 |
| 167 | 0010111 | 1110000000000 |
| 168 | 0100000 | 1000000000000 |
| 169 | 0000100 | 0011000000100 |
| 170 | 0001100 | 0000000100000 |
| 171 | 0100000 | 0010000100000 |
| 172 | 0011000 | 0000000000100 |
| 173 | 0011000 | 1110000000000 |
| 174 | 0100000 | 1000000000000 |
| 175 | 0000101 | 0011000000100 |
| 176 | 0001100 | 0000000100000 |
| 177 | 0100000 | 0010000100000 |
| 178 | 0011001 | 0000000000100 |
| 179 | 0011001 | 1110000000000 |
| 180 | 0100000 | 1000000000000 |
| 181 | 0000110 | 0011000000100 |
| 182 | 0001100 | 0000000100000 |
| 183 | 0100000 | 0010000100000 |
| 184 | 0011010 | 0000000000100 |
| 185 | 0011010 | 1110000000000 |
| 186 | 0011011 | 1000000000000 |
| 187 | 0000000 | 0011000000100 |
| 188 | 0001101 | 0000000100000 |
| 189 | 0010101 | 0010011000100 |
| 190 | 0010101 | 1110000000000 |
| 191 | 0100000 | 1000000000000 |
| 192 | 0000001 | 0011000000100 |
| 193 | 0001101 | 0000000100000 |
| 194 | 0100000 | 0010000100000 |
| 195 | 0010110 | 0000000000100 |
| 196 | 0010110 | 1110000000000 |
| 197 | 0100000 | 1000000000000 |
| 198 | 0000010 | 0011000000100 |
| 199 | 0001101 | 0000000100000 |
| 200 | 0100000 | 0010000100000 |
| 201 | 0010111 | 0000000000100 |
| 202 | 0010111 | 1110000000000 |
| 203 | 0100000 | 1000000000000 |
| 204 | 0000011 | 0011000000100 |
| 205 | 0001101 | 0000000100000 |
| 206 | 0100000 | 0010000100000 |
| 207 | 0011000 | 0000000000100 |
| 208 | 0011000 | 1110000000000 |
| 209 | 0100000 | 1000000000000 |
| 210 | 0000100 | 0011000000100 |
| 211 | 0001101 | 0000000100000 |
| 212 | 0100000 | 0010000100000 |
| 213 | 0011001 | 0000000000100 |
| 214 | 0011001 | 1110000000000 |
| 215 | 0100000 | 1000000000000 |
| 216 | 0000101 | 0011000000100 |
| 217 | 0001101 | 0000000100000 |
| 218 | 0100000 | 0010000100000 |
| 219 | 0011010 | 0000000000100 |
| 220 | 0011010 | 1110000000000 |
| 221 | 0100000 | 1000000000000 |
| 222 | 0000110 | 0011000000100 |
| 223 | 0001101 | 0000000100000 |
| 224 | 0100000 | 0010000100000 |
| 225 | 0011011 | 0000000000100 |
| 226 | 0011011 | 1110000000000 |
| 227 | 0011100 | 1000000000000 |
| 228 | 0000000 | 0011000000100 |
| 229 | 0001110 | 0000000100000 |
| 230 | 0010110 | 0010011000100 |
| 231 | 0010110 | 1110000000000 |
| 232 | 0100000 | 1000000000000 |
| 233 | 0000001 | 0011000000100 |
| 234 | 0001110 | 0000000100000 |
| 235 | 0100000 | 0010000100000 |
| 236 | 0010111 | 0000000000100 |
| 237 | 0010111 | 1110000000000 |
| 238 | 0100000 | 1000000000000 |
| 239 | 0000010 | 0011000000100 |
| 240 | 0001110 | 0000000100000 |
| 241 | 0100000 | 0010000100000 |
| 242 | 0011000 | 0000000000100 |
| 243 | 0011000 | 1110000000000 |
| 244 | 0100000 | 1000000000000 |
| 245 | 0000011 | 0011000000100 |
| 246 | 0001110 | 0000000100000 |
| 247 | 0100000 | 0010000100000 |
| 248 | 0011001 | 0000000000100 |
| 249 | 0011001 | 1110000000000 |
| 250 | 0100000 | 1000000000000 |
| 251 | 0000100 | 0011000000100 |
| 252 | 0001110 | 0000000100000 |
| 253 | 0100000 | 0010000100000 |
| 254 | 0011010 | 0000000000100 |
| 255 | 0011010 | 1110000000000 |
| 256 | 0100000 | 1000000000000 |
| 257 | 0000101 | 0011000000100 |
| 258 | 0001110 | 0000000100000 |
| 259 | 0100000 | 0010000100000 |
| 260 | 0011011 | 0000000000100 |
| 261 | 0011011 | 1110000000000 |
| 262 | 0100000 | 1000000000000 |
| 263 | 0000110 | 0011000000100 |
| 264 | 0001110 | 0000000100000 |
| 265 | 0100000 | 0010000100000 |
| 266 | 0011100 | 0000000000100 |
| 267 | 0011100 | 1110000000000 |
| 268 | 0011101 | 1000000000000 |
| 269 | 0000111 | 0000000000100 |
| 270 | 0001111 | 0011000100000 |
| 271 | 0011110 | 1100000000000 |

) FIM DA INTERPRETACAO - Nenhum erro detectado

## REFERÊNCIAS BIBLIOGRÁFICAS


[AGG 87] AGGARWAL J. K. & LEE S. Y. A mapping strategy for parallel processing. IEEE Transactions on Computer, New York, C-36(4):207-22, Apr. 1987.

[BHU 87] BHUYAN, L. N. Interconnection networks for parallel and distributed processing. Computer. New York, 20(6):9 12, June 1987.

[BOK 81] BOKHARI S. H. On the mapping problem. IEEE Transactions on Computer, New York, C-30(3):207-22, Mar. 1981.

[BOR 87] BORLAND International. TURBO C user's guide. Scott Valley, Borland International, 1987.

[CAP 85] CAPPELLO P. R. A mesh automaton for solve dense linear systems. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 20-23, 1985. Proceedings. New York, IEEE, 1985. p.418-25.

[CAR 85] CARLSON D. A. Performing tree and prefix computations on modified mesh-connected parallel computer. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 20-23, 1985. Proceedings. New York, IEEE, 1985. p. 715-18.

[CHI 83] CHIANG Y. P. & FU K. S. Matching parallel algorithm and architecture. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Bellaire, Aug. 23-26, 1983. Proceedings. Silver Spring, IEEE, 1983. p.374-80.

[CRA 68] CRANE B. A. Path finding with associative memory. IEEE Transactions on Computer, New York, 17(7):691-93, July 1968.

[CLI 85] CLINE C. L. & SIEGEL H. J. Augmenting ADA for SIMD processing. IEEE Software Enginnering, New York, 11(9):207-22, Sept. 1985.

[DUD 72] DUDA R. O. & HART P. E. Use of the Hough transformation to detect lines and curves in pictures. Communications of ACM, New York, 15(1):11-15, Jan. 1972.

[FEN 72] FENG T. Y. Some characteristics of associative parallel processing. In: SAGAMORE COMPUTER CONFERENCE ON PARALLEL PROCESSING, Syracuse. Proceedings. 1972. p. 5-12.

[FEN 81] FENG T. Y. A survey of interconnection networks. Computer, New York, 12(6):12-27, Dec. 1981.

[FLA 82] FLANDERS P. M. A unified approach to a class of data movements on array processor. IEEE Transactions on Computer, New York, C-31(9):809-19, Sept. 1982.

[FLY 72] FLYNN M. J. Some computer organizations and their effectiveness. IEEE Transactions Computer, New York, C-21(9):948-60, Sept. 1972.

[GIL 71] GILMORE P. A. Numerical solution of partial differential equations by associative processing. In: FALL JOINT COMPUTER CONFERENCE, Las Vegas, Nov. 16-18,1971. Proceedings. Montvale, AFIPS press, 1971. p.411-18.

[GOP 85] GOPALAKRISHNAN P. S. RAMAKRISHNAN I. V., KANAL L. N. An efficient connect components algorithm on a mesh connected computer. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 20 23, 1985. Proceedings. New York, IEEE, 1985. p. 711-14.

[HAN 77] HANDLER W. The impact of classification schemes on computer architecture. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Michigan, Aug. 23-26, 1977. Proceedings. New York, IEEE, 1972. p.7-15.

[HON 79] HON R. & W. REDDY D. R. The effect of computer archicteture on algorithm decomposition and performance. In: High speed computer and algorithm organization. New York, Academic Press, 1979, p. 411-21.

[HWA 84] HWANG K. & BRIGGS F. Computer architecture and Parallel Processing. New York, McGraw-Hill,1984.

[KUE 85] KUEHN J. T. & SIEGEL H. J. Extensions to the C programming language for SIMD/MIMD parallelism. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 22-23, 1985. Proceedings. New York, IEEE, 1985, p. 719-26.

[KUN 80] KUNG H. T. The structure of Parallel Algorithms. Advances in Computer, New York, 19:65-112, 1986.

[KUN 82] KUNG H. T. Why systolic architectures ?. IEEE Computer, New York, 15(1):37-46, Jan. 1982.

[KUN 88] KUNG S. Y. VLSI array processors: designs and applications. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, Espoo, June 7-9, 1988. Proceedings. New York, IEEE, 1988. v.1 p. 313-20.

[LAW 75] LAWRIE D. H. et al. GLYPNIR - A programming language for the ILLIAC IV. Communication of the ACM, New York, 18(3):157-164, Mar. 1975.

[INT 81] INTEL Corporation MCS-51 Family of single-chip microcomputers User's Manual. 1981.

[LEE 63] LEE C. & PAUL M. C. A content addressable whit distribuited logic memory to information retrieval. Proceedings of IEEE, New York, 1(7):942-32, June, 1963.

[LIN 85] LIN T. & MOLDOVAN D. Tradeoffs in mapping algorithms to array processor. in: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 20-23. Proceedings. New York, IEEE, 1985. p. 719-26.

[MAS 84] MASCARENHAS N. & VELASCO F. Processamento digital de imagens. São Paulo, USP, 1984.

[MOL 83] MOLDOVAN D. On the design of algoritms for VLSI systolic array. Proceedings of IEEE, New York,71(1):113-20, Jan. 1983.

[NCR 85a] NCR Corporation. GAPP Personal computer development system user's manual. Dayton, NCR, 1985.

[NCR 85b] NCR Corporation. Detection of edges and gradients in binary and gray scale images with the GAPP processor. Dayton, 1985. GAPP application note nro. 3.

[NCR 85c] NCR Corporation. Data input/output techniques for the geometric arithmetic parallel processor (GAPP). Dayton, 1985. GAPP application note nro. 5.

[NCR 85d] NCR Corporation. Geometric Arithmetic Parallel Processor. Dayton, 1985. NCR45CG72.

[OFF 85] OFFEN R. J. VLSI Image Processing. London, Willian Collins, 1985.

[ORL 72] ORLANDO V. A. & BERRA P. B. The solution of the minimum cost flow and maximum flow networks problems using associative processing. In: FALL JOINT COMPUTER CONFERENCE, Anahein, Dec. 5-7, 1972. Proceedings. Montvale, AFIPS press, 1972. p. 859-66.

[PAI 85] PAIGE R. C. & KRUSKAL C. P. Parallel algorithms for shortest path problems. in: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St. Charles, Aug. 20-23, 1985. Proceedings. New York, IEEE, 1985. p. 14-19.

[PAR 73] PARHAMI B. Associative memories and processor: An overview and selected bibliography. Proceedings of IEEE. New York, 61(6):722-30, June 1973.

[PER 79] PERROT R. H. A language for array e vector processing. ACM Transaction on Programming Languages and Systems, New York, 1(2):177-95, Oct. 1975.

[PRA 78] PRAT W. K. Digital image processing. New York, John Wiley, 1978.

[RAB 75] RABINER L. & GOLD B. _Theory and application of digital signal processing_. New Jersey, Prentice-Hall, 1975.

[SHO 60] SHOMAN W. Parallel computing with vertical data. In: EASTERN JOINT COMPUTER CONFERENCE. _Proceedings_. New York, 1960, p. 111-15.

[STE 75] STEVENS K. G. CFD - A FORTRAN like language for the Illiac IV. _SIGPLAN notices_, New York, _10_(3):72-6, Mar. 1975.

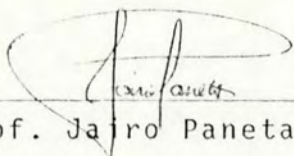
[THU 74] THURBER. K. J. Interconnection networks - a survey and assessment. In: NATIONAL COMPUTER CONFERENCE AND EXPOSITION, Chicago, May 6-10, 1974. _Proceedings_. Montvale, AFIPS Press, 1974. p.909-19.

[VOI 85] VOIGHT R. G. Where are the parallel algorithms? In: NATIONAL COMPUTER CONFERENCE, Chicago, July 15-18, 1985. _Proceedings_. Montvale, AFIPS Press, 1979. p.329-34.

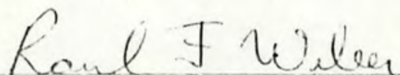
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Pós-Graduação em Ciência da Computação

Implementação de Arquiteturas SIMD

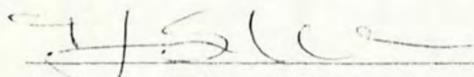
Dissertação apresentada aos Srs.



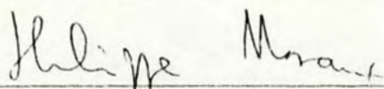
Prof. Jairo Paneta



Prof. Raul Fernando Weber

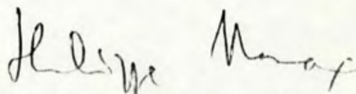


Profa. Taisy Silva Weber

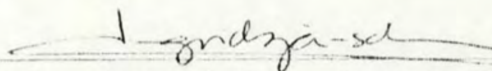


Prof. Philippe Olivier A. Navaux

Visto e permitida a impressão
Porto Alegre, 04./09./89.



Prof. Philippe O. A. Navaux
Orientador



Profa. Ingrid E.S. Jansch Pôrto
Coordenadora do Curso de Pós-Graduação
em Ciência da Computação