Um Ambiente de Programação e Depuração para um Processador SIMD

Maria Fernanda Eppinghaus Belford Roxo[1]

Adriano Joaquim de Oliveira Cruz[2]

Maurício Abud Seabra da Cruz[3]

Resumo

Este trabalho apresenta uma visão geral do ambiente desenvolvido para programação e depuração de programas paralelos para o projeto PRIMATA, que é uma pesquisa em arquiteturas SIMD. O ambiente desenvolvido usando-se C, X-Windows e Motif, é portável, fácil de instalar e usar. Usuários podem controlar e observar quase todos os recursos da máquina. Uma linguagem assembly parecida com as usadas nos microprocessadores mais comuns está disponível.

Abstract

This work presents an overview of the environment developed for programming and debugging parallel programs for the PRIMATA project, which is a research on SIMD architectures. The environment, developed using C, X-Windows and Motif, is portable, easy to install and use. Users can watch and control almost every resource of the machine. An assembly language very similar to the ones use in everyday microprocessors is also available.

[1] M.Sc COPPE/UFRJ; Pesquisadora NCE/UFRJ; E-mail:fernanda@nce.ufrj.br
Endereço:NCE/UFRJ, Cx.Postal 2324, Rio de Janeiro, RJ, CEP-20001-970

[2] Ph.D University of Southampton; Pesquisador NCE/UFRJ; Professor IM/UFRJ;
E-mail:adriano@nce.ufrj.br

[3]Bolsista do CNPq; E-mail:abud@magma.leg.ufrj.br

1. Introdução

O PRIMATA (Processador de Imagens e Matrizes) é uma arquitetura maciçamente paralela operando em modo SIMD e orientada para processamento de imagens e matrizes. A principal unidade do PRIMATA é uma matriz de elementos processadores (EPs) de quatro bits cada uma. Em arquiteturas SIMD [Red73, Bat80, Hil85, Jes87, Bla90], o desempenho depende basicamente do produto do número de EPs pela velocidade individual de cada EP. O importante é manter um compromisso entre desempenho e custo a fim de viabilizar o número de EPs. As pesquisas realizadas durante o desenvolvimento do projeto resultaram em um EP de alta eficiência como está demonstrado em [Rox94].

Algumas das vantagens deste tipo de arquitetura são o baixo custo individual de cada EP, a fácil migração de programas seqüenciais já existentes e a excelente adaptação às aplicações para as quais o computador será usado. A fácil migração pode ser explicada pelo fato das máquinas SIMD poderem ser consideradas como uma implementação em hardware de dois laços de programa aninhados, como por exemplo:

```
for ( i = 0; i < n; i++ )
  for ( j = 0; j < n; j++ )
        a[i,j] = b[i,j] + c[i,j];
```

Este trabalho apresenta um ambiente de programação e depuração de programas paralelos segundo o modelo SIMD que foi desenvolvido para o computador PRIMATA [Rox93, Rox94]. Criado de maneira a ser portável, ele pode ser utilizado como ferramenta de pesquisa nesta área por centros que disponham de estações de trabalho com X-Windows e Motif. O ambiente permite a programação do PRIMATA em uma linguagem assembly especialmente desenvolvida para o projeto. Apesar de ser um computador, paralelo este assembly é muito similar aos usados nos microprocessadores atuais. O depurador embutido no ambiente permite a visualização de praticamente todos os recursos da máquina e controle total da execução dos programas.

2. Descrição da Arquitetura

Para apresentar o ambiente de programação e simulação do PRIMATA optou-se por descrever a arquitetura do ponto de vista do simulador. Uma descrição mais detalhada da arquitetura pode ser encontrada em [Rox94]. A arquitetura do PRIMATA é composta basicamente por uma Unidade de Controle (UC), uma memória para dados e programa associada a UC (com largura de 32 *bits*) e uma Matriz de Elementos Processadores (MEP).

A Unidade de Controle é responsável por buscar as instruções do programa na memória, decodificá-las, executar as "instruções escalares" e enviar o controle das "instruções paralelas" para a MEP, além de calcular o endereço da próxima instrução. As instruções escalares são aquelas que afetam exclusivamente os registradores da UC ou o fluxo de instruções. Já as instruções paralelas são aquelas executadas por todos os Elementos Processadores (EPs). A UC é ainda responsável pelo endereçamento do banco de memórias associado à MEP. Normalmente, uma instrução de programa contém uma parte escalar e outra paralela, como será visto mais adiante.

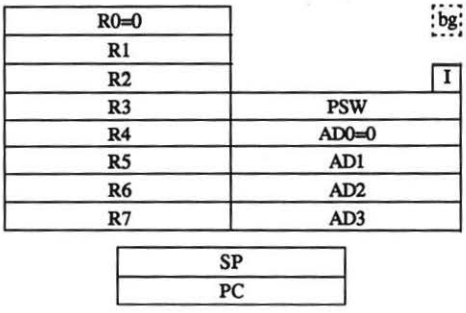

Figura 1 - Modelo de Programação da Unidade de Controle

A Figura 1 apresenta o modelo de programação da Unidade de Controle. Com exceção do registrador I, todos os demais registradores têm 32 *bits*. O registrador PC (*Program Counter*) aponta para a próxima instrução, enquanto que o registrador SP (*Stack Pointer*) aponta para o topo da pilha. Os registradores Rx (R0, R1, R2, R3, R4, R5, R6 e R7) são registradores de uso geral, sendo que o registrador R0 tem valor fixo igual a zero. Os registradores ADx (AD0, AD1, AD2 e AD3) são utilizados para endereçar a memória da MEP, sendo que o registrador AD0 tem valor fixo igual zero. O registrador PSW (*Program Status Word*) armazena o valor atual das *flags* da UC. PSW possui apenas cinco *flags*: ZF (*zero flag*), CF (*carry flag*), SF (*sign flag*), OF (*overflow flag*) e Sum-OR (*bit* de controle enviado pala matriz de elementos processadores), sendo que os quatro primeiros dependem exclusivamente do resultado das operações escalares. O registrador I possui apenas 3 *bits* e é utilizado exclusivamente para indexar o banco de registradores A dos Elementos Processadores (que será apresentado mais adiante). Por último, bg não é propriamente um registrador, mas o conteúdo do barramento global, que é uma palavra de 4 *bits*, usada para transmitir constantes a todos os EPs ao mesmo tempo.

A Matriz de Elementos Processadores reúne um conjunto de EPs dispostos espacialmente na forma de uma matriz bidimensional. A comunicação de dados entre os

EPs da matriz é processada por uma rede de interconexões em X de 1 *bit*, como mostra a Figura 2. Cada EP está associado a quatro linhas direcionadas aos seus vizinhos diagonais: NE, SE, NO e SO. Na interseção das linhas que se cruzam há uma conexão física. Isto permite que cada EP se comunique com seus 8 vizinhos mais próximos: norte (N), nordeste (NE), este (E), sudeste (SE), sul (S), sudoeste (SO), oeste (O) e noroeste (NO) utilizando apenas 4 linhas. O fluxo da comunicação de dados é definido pela combinação das direções de envio e recebimento de dados. Por exemplo, para estabelecer um fluxo de dados na direção norte, basta que cada EP envie dados na direção nordeste (NE) e receba dados da direção sudeste (SE).

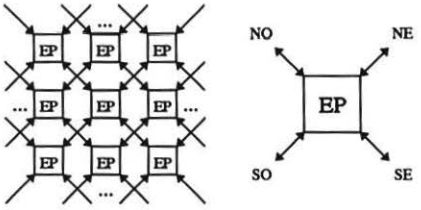

Figura 2 - Rede de interconexões entre EPs.

A Figura 3 apresenta um diagrama funcional do Elemento Processador do PRIMATA. Cada EP contém: dois registradores de 1 *bit* (K e C), um registrador de 4 *bits* (B), um registrador de deslocamento de 4 *bits* (SH) e um banco de registradores (A). O banco de registradores A é composto por oito registradores de deslocamento de 4 *bits* interligados, formando um registrador de deslocamento de 32 *bits*. Cada EP está ainda associado a um memória RAM (M). Para auxiliar a execução das operações lógicas e aritméticas são utilizados: uma unidade lógica de 1 *bit* (LK), uma unidade lógica e aritmética de 4 *bits* (ULA) e um operador lógico OR (que testa se o barramento b4 é igual a zero). Cada EP possui dois barramentos principais: o barramento de 1 *bit* (b1) e o barramento de 4 *bits* (b4).

Pode-se dividir a arquitetura do EP em dois blocos principais: um bloco de 1 *bit* e um bloco de 4 *bits*. O processador pode executar operações de 1 e 4 *bits* em paralelo. Os elementos responsáveis pela comunicação de dados entre estes dois blocos são: o registrador de deslocamento SH, o operador OR e uma ligação direta entre o *bit* mais significativo do barramento b4 (*bit* 3) e o barramento b1.

O registrador C é utilizado para armazenar o *carry* das operações aritméticas. Ele pode ser carregado com '1' (um), '0' (zero), o conteúdo de b1, ou com a saída de *carry* da ULA. A ULA pode executar uma operação lógica entre: o registrador B e o registrador K, ou entre o registrador B e um valor lido do banco de registradores A. A

ULA pode ainda executar a soma entre: o resultado desta operação lógica, o registrador de *carry* C e, opcionalmente, um valor lido do banco de registradores A.

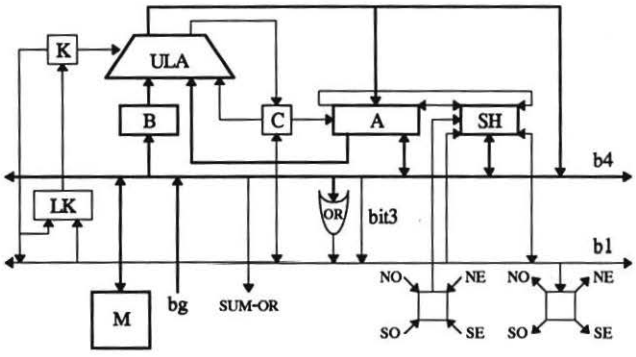

Figura 3 - Elemento Processador

O registrador B é utilizado para armazenar um dos operandos de uma operação lógica ou aritmética. O outro operando é armazenado no banco de registradores A. Este banco de registradores tem a função de uma memória interna, com a vantagem de permitir o deslocamento bidirecional de dados, o que facilita enormemente a execução de algumas operações aritméticas, como a multiplicação e a divisão, por exemplo. Apenas as operações de deslocamento (para a esquerda e para a direita) são executadas simultaneamente em todos os registradores que compõem o banco. As operações de escrita e leitura são executadas em apenas um registrador de cada vez. Para selecionar qual registrador do banco será lido ou escrito é utilizado o registrador I da UC.

O registrador K é chamado de registrador de máscara. Este registrador é uma das características de arquiteturas SIMD e permite o controle individual de cada processador. Através do registrador de máscara é possível implementar de forma simples uma estrutura do tipo SE <condição> FAÇA <expressão>. Embora esta estrutura pareça simples em arquiteturas seriais, em uma arquitetura SIMD ela não seria trivial sem o auxílio de um registrador de máscara, já que todos os processadores executam a mesma instrução. Para implementá-la, inicialmente o resultado da condição é armazenado no registrador K. Depois a expressão é executada utilizando operações condicionais a K. Todas as operações que alteram o conteúdo do banco de registradores A podem ser condicionais a K. Além disto, no caso do PRIMATA, como o registrador K pode ser um dos operandos da ULA, pode-se escolher uma operação lógica que torne a expressão condicional a K. Neste caso, pode-se ainda implementar estruturas do tipo SE <condição> FAÇA <expressão1> NO_RESTANTE <expressão2>, executando as

expressões 1 e 2 em paralelo, simplesmente através da escolha apropriada da operação lógica. Por exemplo, para executar a expressão (A - B) caso uma condição seja satisfeita e (A + B) caso contrário, basta armazenar o resultado da condição nos registradores K e C e depois executar a expressão [A + (B xor K) + C].

Uma outra característica das arquiteturas SIMD é árvore SUM-OR. Cada EP envia um *bit* para a UC. Todos estes *bits* passam por uma árvore de portas lógicas OU, de forma que a UC receba apenas 1 *bit*, resultado da operação lógica OU de todos os *bits* enviados. Este *bit* é o único mecanismo que a UC tem para tomar decisões baseada no estado dos EPs.

3. Ambiente de Simulação

O ambiente de simulação foi desenvolvido em estações SUN, usando a linguagem C e uma interface gráfica X-Windows/MOTIF. Procurou-se criar um ambiente com as facilidades oferecidas por um compilador e um depurador. A maior diferença é que os resultados apresentados não são variáveis do programa, mas o conteúdo de registradores e outros elementos da arquitetura. A janela principal da interface é apresentada na figura 4 e pode ser dividida em seis áreas: barra de menu, arquivos fontes, botões da arquitetura, área de resultados, área de programa e botões de controle de fluxo. A barra de menu apresenta opções para salvar e recuperar o contexto de simulação (programa fonte, arquivo de dados, elementos visíveis e resultados obtidos). Ela contém ainda opções de configuração do simulador.

Existem duas pequenas linhas de texto contendo o nome do programa fonte e o nome do arquivo de dados. Ao entrar com o nome de um novo programa fonte, o simulador automaticamente lê o arquivo, compila e apresenta os erros, caso eles existam. Se o programa fonte estiver correto, um programa objeto é gerado e carregado na memória da UC. Além disto o programa fonte é carregado na área de programa. A área de programa destaca a instrução atual e os pontos de parada (*breakpoints*) selecionados.

Através do arquivo de dados é possível carregar valores na memória dos EPs, os seja, os dados a serem processados pelo programa. Cada linha do arquivo de dados contém: a linha e a coluna que definem qual EP terá o conteúdo da sua memória alterado, o endereço a partir do qual os dados serão carregados e os valores propriamente ditos. Há ainda a opção de carregar os mesmos dados em uma determinada linha ou coluna da MEP, ou até mesmo em todos os EPs.

A área contendo botões com os elementos da arquitetura define os elementos cujos valores serão apresentados na área de resultados ao longo da simulação, denominados

elementos visíveis. Caso o botão da Unidade de Controle seja selecionado, o conteúdo de todos os registradores da mesma é apresentado. A MEP tem um controle mais preciso. Pode-se escolher que EPs serão apresentados e ainda que elementos dos mesmos serão apresentados. Se nenhum for selecionado, o valor de todos os elementos dos EPs selecionados são apresentados. Todos os resultados da simulação são armazenados e ficam disponíveis na área de resultados para análise.

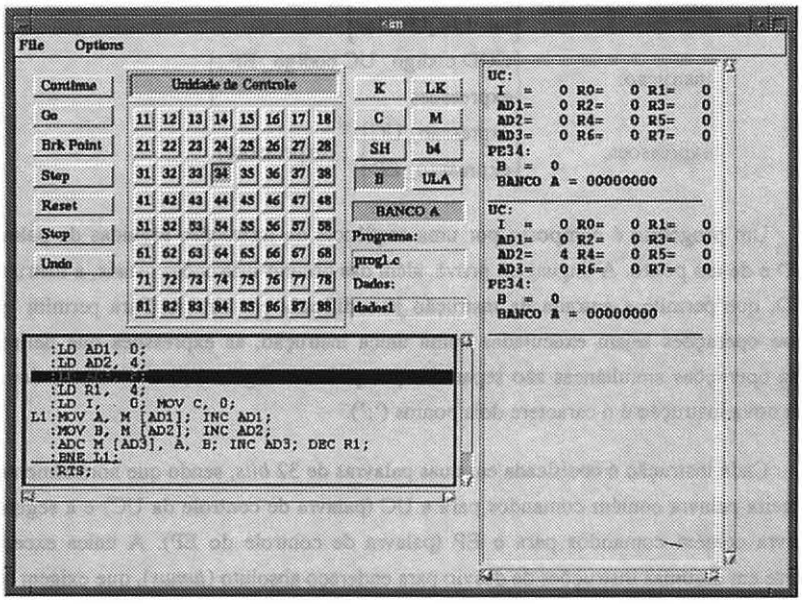

Figura 4 - Janela principal do Simulador

Os botões de controle de fluxo permitem que o programa seja executado passo a passo (Step), executado até o próximo ponto de parada (Continue), executado até o fim (Go) ou até que haja uma interrupção por parte do usuário (Go/Stop). Eles permitem ainda marcar pontos de parada (BrkPoint), reiniciar o programa (Reset) e voltar ao contexto do último resultado apresentado (Undo).

4. Ambiente de Programação

A linguagem assembly desenvolvida para o PRIMATA assemelha-se à de um dos microprocessadores atuais, porém com algumas características especiais, decorrentes do paralelismo da arquitetura. É importante que a linguagem explore ao máximo o paralelismo. Sendo assim, numa única instrução são permitidas várias operações, tanto na Unidade de Controle quanto nos Elementos Processadores.

A estrutura básica de um programa é apresentada abaixo. Os colchetes significam expressões opcionais e as chaves representam opções de expressões.

programa: [instrucoes] END.

instrucoes: { pseudo_instrucao / [label] : instrucao } [instrucoes]

pseudo_instrucao: { ORG endereco / label EQU valor } ;

instrucao: { COD codigo_UC codigo_EP ; / expressoes }

expressoes: { expressao_UC / expressao_EP } ; [expressoes]

Um programa é composto por uma seqüência de instruções seguidas da palavra END e de um ponto. A linguagem provê, além das pseudo-instruções usuais, a instrução COD, que permite a entrada da instrução já codificada em binário. Para permitir que várias operações sejam executadas numa única instrução, as expressões que definem estas operações simultâneas são separadas por ponto-e-vírgula (';') e o delimitador de uma nova instrução é o caractere dois pontos (':').

Cada instrução é codificada em duas palavras de 32 *bits*, sendo que normalmente a primeira palavra contém comandos para a UC (palavra de controle da UC) e a segunda palavra contém comandos para o EP (palavra de controle do EP). A única exceção ocorre em algumas instruções de desvio para endereço absoluto (*jumps*), que exigem que a segunda palavra seja usada para armazenar o endereço de desvio. Sendo assim, para estas instruções não são permitidos comandos para o EP.

A escolha dos *bits* que definem as duas palavras de controle levou em consideração as operações que podem ser executadas em paralelo e as instruções mais usadas. Desta forma, cinco *bits* da palavra de controle da UC foram reservados para operações específicas: definição do registrador ADx utilizado para endereçar a memória dos EPs, pós-incremento de um registrador ADx e pós-incremento do registrador I. Os demais *bits* são utilizados para definir o *opcode* de outra operação da UC e seus operandos. Já a palavra de controle do EP permite que quase todas as operações que podem ser processadas em paralelo possam estar definidas numa única instrução. A única exceção se refere a transferência de dados entre os EPs, a qual não pode ser processada na mesma instrução que uma operação lógica envolvendo a unidade lógica LK. Todas estas possibilidades de paralelismo tornam a instrução bastante poderosa.

As operações da UC se assemelham às instruções do assembly de um processador RISC. Elas podem ser divididas em quatro tipos: operações de carregamento, operações lógicas e aritméticas, operações com pilha e operações de desvio. As operações de carregamento permitem a transferência de dados entre um registrador Rx e a memória, entre dois registradores Rx e a atribuição de valores a um registrador qualquer. As operações lógicas e aritméticas podem ter um, dois, ou três operandos. A UC trabalha com três tipos de dados: *word* (32 *bits*), *halfword* (16 *bits*) e *byte* (8 *bits*). Embora a memória da UC tenha largura de 32 bits, ela é endereçada *byte* a *byte*. Sendo assim, para ter acesso a *words* subsequentes da memória é preciso incrementar o endereço de quatro unidades. São providas instruções de leitura e escrita na memória dos três tipos de dados, além de transformação dos tipos menores para os tipos maiores através da extensão do sinal do dado. LD é a instrução de carregamento de 32 *bits*. As letras S e U indicam operações com e sem sinal, enquanto que B e H se referem a *bytes* e *halfwords*, respectivamente.

TIPO	OPERAÇÕES	EXEMPLOS
Leitura/Escrita da memória	LD, LDSB, LDUB, LDSH, LDUH ST, STB, STH	LD R1, (R2 + 12) ST R5, (R3 - 4)
Atribuição de valor imediato	LD, LDS, LDHI	LDS AD1, -17
Transferência entre registradores	LD	LD R7, R4
Aritméticas com 1 operando	INC, DEC	INC I
Deslocamentos Lógicos e Aritméticos com 2 operandos	SHL, SHR, ASHL, ASHR, ROL, ROR, ROLC, RORC, SEX, SEXB	SHR R5, R3
Lógicas e Aritméticas com 3 operandos	ADD, ADC, SUB, SBC, AND, OR, XOR	OR R5, R3, R1 ADD AD2, R2, -14
Pilha	LD PUSH, POP	LD R7, (SP + 4) PUSH PSW
Desvio	Bcc, BRA, BSR (relativos) Jcc, JMP, JSR (absolutos)	BSR label_1 JMP R4
Retorno de subrotina	RTS	RTS

Tabela 1 - Instruções da Unidade de Controle

As operações com pilha permitem não só acrescentar e retirar valores na pilha, como também ler dados da pilha sem ter que excluí-los da mesma. São providas instruções de desvio para um endereço relativo (*branch*) ou absoluto (*jump*). Os desvios podem ser condicionais ou incondicionais, porém as chamadas de subrotina são sempre incondicionais. A Tabela 1 apresenta um resumo dos diversos tipos de operações permitidos para a UC.

As operações do EP definem basicamente as funções executas por LK e pela ULA, as entradas dos barramentos b1 e b4, o carregamento ou não de novos valores nos

registradores e a direção da transferência de dados entre EPs (quando for o caso). Essas definições podem estar numa única expressão ou em expressões diferentes. É possível a existência de mais de uma maneira de definir uma determinada operação ou conjunto de operações. Por exemplo, para somar os registradores A, B e C, armazenando o resultado em SH e o *carry* resultante em C, pode-se usar as instruções equivalentes abaixo:

```
•:ADC SH, A, B
•:ADD SH, A, B; MOV C, C4
•:MOV SH, A + B; MOV C, C4
```

Observe que a operação ADC, ao contrário do usual, não significa que o registrador de *carry* (C) é somado aos outros dois operandos, o que ocorre em todas as operações de soma nesta arquitetura, mas sim que o *carry* resultante da soma é armazenado no registrador C. Observe ainda que C4 é uma palavra reservada criada para designar a saída de *carry* da ULA. Foi criado um registrador lógico ASH, que corresponde a concatenação do banco de registradores A com o registrador SH, criado para facilitar as operações de deslocamento que envolvem A e SH simultaneamente (Figura 3).

TIPO	OPERAÇÃO	1° operando	2° operando	3° operando
Soma	ADD ou ADC	A, B, SH, ou M[ADx]	0 ou A	lógica_ULA
Carregamento	MOV	A, B, SH, ou M[ADx]	b4	
	MOV	C	0, 1, b4, ou C4	
	MOV	K	lógica_LK	
Deslocamento	SHL, SHR, ou ROL	A ou ASH		
	ROR	A ou SH		
	SHL ou SHR	SH	A ou b1	
Transferência	TRANSF	N, NE, E, SE, S, SO, O, ou NO	b1	

Tabela 2 - Resumo das operações do EP e seus operandos

Um resumo das principais operações permitidas para o EP é apresentado na Tabela 2. Para esta tabela, lógica_ULA é qualquer operação lógica entre A e B ou entre B e K. Da mesma forma, lógica_LK é qualquer operação lógica entre K e b1. Além disto, b1 é qualquer entrada do barramento b1 (K, C, LSB SH, MSB b4, ou SUM b4), ou a própria palavra reservada B1, correspondendo a entrada b1 já definida em outra operação da mesma instrução. Da mesma forma, b4 é qualquer entrada do barramento b4 (SH, M[ADx], BG, lógica_LK, 0+lógica_LK, ou A+lógica_LK), ou a palavra reservada B4. Embora não constem desta tabela, todas as operações que alteram o valor

de A podem ser condicionais ao registrador de máscara K. As operações condicionais são precedidas da letra C.

5. Exemplos

Para melhor compreender a linguagem apresentada e os problemas que envolvem a programação de uma arquitetura SIMD, serão apresentados dois exemplos de algoritmos simples implementados no ambiente apresentado. O primeiro é a soma de duas matrizes e o segundo é o cálculo do valor máximo de uma matriz. Para facilitar a compreensão, será considerado que as dimensões das matrizes a serem processadas são as mesmas da Matriz de Elementos Processadores. Ou seja, cada EP armazena o valor de um elemento de cada matriz processada.

5.1 - Soma

Considere a soma de duas matrizes X e Y, tendo como resultado a matriz Z, onde as matrizes têm dimensão n X n e elementos inteiros de 16 *bits*. Num processamento escalar, a soma dessas duas matrizes seria processada em n^2 passos. Numa arquitetura SIMD, a soma de todos os elementos da matriz é processada simultaneamente, com cada EP somando um elemento da matriz. Como o EP do PRIMATA é um processador de 4 *bits*, a soma terá que ser quebrada em 4 subsomas de 4 *bits* cada. Ou seja, a soma das duas matrizes será processada em 4 passos. A seguir é apresentado o programa fonte que implementa a soma da matrizes X (com endereço inicial 0) e Y (com endereço inicial 4),.com resultado armazenado na matriz Z (com endereço inicial 8).

```
1       :LD AD1, 0;
2       :LD AD2, 4;
3       :LD AD3, 8;
4       :LD R1, 4;
5       :LD I, 0; MOV C, 0;
6   L1  :MOV A, M [AD1]; INC AD1;
7       :MOV B, M [AD2]; INC AD2;
8       :ADC M [AD3], A, B; INC AD3; DEC R1;
9       :BNE L1;
10      :RTS;
```

As três primeiras instruções carregam as bases de endereço das três matrizes nos registradores AD1, AD2 e AD3. A quarta instrução carrega em R1 o número de passos da soma (4). O registrador R1 é utilizado para controlar o *loop* do programa. A quinta instrução inicia o registrador de *carry* C e o índice I do banco de registradores A com

zero. A sexta, sétima e oitava instruções compõem o *loop* que executa a subsoma de 4 *bits*. A sexta instrução carrega em A 4 *bits* de um elemento da matriz X, incrementando ao final da instrução o endereço de acesso à matriz X, para ter acesso aos próximos 4 *bits* do elemento na próxima iteração. Da mesma forma, a sétima instrução carrega em B parte de um elemento da matriz Y, incrementando a seguir o endereço de acesso da matriz Y. A oitava instrução soma os valores armazenados em A, B e C, escrevendo o resultado na matriz Z e atualizando o conteúdo do registrador de *carry* C. Esta instrução ainda incrementa o endereço de acesso à matriz Z e decrementa o contador do *loop* armazenado em R1. A nona instrução testa o fim do *loop* com a operação BNE (*Branch if not equal*), desviando o fluxo do programa para a sexta instrução para uma nova iteração caso R1 seja maior que zero. No fim do *loop* o programa executa a instrução RTS que encerra o programa.

5.2. Cálculo do Máximo

Considere o cálculo do elemento de maior valor de uma matriz X com dimensão n X n e elementos inteiros positivos de 16 *bits*. Num processamento escalar, este cálculo seria processado em n^2 passos. Numa arquitetura SIMD, a melhor opção é comparar simultaneamente todos os elementos da matriz, *bit* a *bit*. Com isto, o máximo pode ser calculado em apenas 16 passos. O algoritmo baseia-se na árvore Sum-OR. Inicialmente os registradores mais altos do banco de registradores A são carregados com os elementos da matriz e o registrador B é carregado com o valor 15 (1111 em binário), habilitando a atividade de todos os EPs. Neste algoritmo ao invés de usar o registrador de máscara para inibir a atividade dos EPs, por uma questão de otimização do algoritmo, escolheu-se o *bit* mais significativo do registrador B para indicar a atividade do EP. O resultado é armazenado no registrador de uso geral R2, que inicialmente é carregado com zero. Os 16 *bits* mais significativos do banco de registradores A são então testados, *bit* a *bit* (do mais significativo para o menos significativo) e seu valor enviado para a árvore Sum-OR, caso o EP esteja ativo. Se o resultado da árvore Sum-OR for igual a um, significa que pelo menos um EP ativo tem o *bit* testado igual a um e o seu elemento é maior do que o dos EPs ativos que têm o mesmo *bit* igual a zero. Neste caso, desativa-se os EPs que têm o *bit* igual a zero, desloca-se o conteúdo de registrador R2 uma posição pará a esquerda e armazena-se 1 no *bit* menos significativo de R2. Se o resultado da árvore Sum-OR for igual a zero, significa que todos os EPs ativos tem o *bit* testado igual a zero. Logo, apenas desloca-se o conteúdo de registrador R2 uma posição para a esquerda, tornando o seu *bit* menos significativo igual a zero, sem desativar nenhum EP. Ao final de 16 iterações o resultado estará armazenado no registrador R2. A

seguir é apresentado o programa fonte que implementa o algoritmo descrito. Considere que o endereço inicial da matriz testada é igual a zero.

```
1       :LD AD1, 0;
2       :LD I, 4;
3       :LD R1, 3;
4   L1  :MOV A, M[AD1]; INC AD1; INC I; DEC R1;
5       :BNE L1;
6       :MOV A, M[AD1]; LD R1, 16;
7       :MOV B, 15; LD R2, 0;
8   L2  :MOV K, MSB B AND A; SHL A; SHL R2, R2;
9       :BNS L3;
10      :MOV B, B AND K; INC R2;
11  L3  :DEC R1;
12      :BNE L2;
13      :RTS;
```

As três primeiras instruções iniciam os valores da base de endereço da matriz (AD1), do indexador (I) do banco de registradores A e do contador (R1) do *loop* de carregamento do valor elemento no banco de registradores A. A quarta, quinta e sexta instruções implementam o *loop* de carregamento do elemento da matriz em A. O elemento é carregado nas posições 4, 5, 6 e 7 do banco A. A cada iteração, um palavra de 4 *bits* do elemento é carregada na posição I do banco A e há um pós incremento da base de endereço do elemento e do índice I. Há ainda um decremento do contador do *loop*. A última iteração (sexta instrução) encontra-se separada para que o índice I não seja incrementado, mas sim apontando para a palavra mais significativa até o final do programa. Nesta instrução é ainda iniciado o contador (R1) do *loop* de cálculo do máximo. A sétima instrução inicia o registrador B com 15, habilitando a atividade de todos os EPs e o resultado (R2) com zero. A oitava instrução testa o *bit* mais significativo de A, enviando o mesmo para a árvore Sum-OR caso o EP esteja ativo (*bit* mais significativo de B igual a um) ou enviando zero caso o EP esteja inativo. O resultado do teste é ainda armazenado no registrador de máscara K. Nesta mesma instrução, o banco A é deslocado de uma posição para a esquerda, a fim de prepará-lo para o teste do próximo *bit* e o registrador R2 também é deslocado de uma posição para a esquerda, a fim de prepará-lo para armazenar o próximo *bit* do resultado. A nona instrução (BNS - *branch if not Sum-OR*) testa se o resultado da árvore Sum-OR é igual a zero. Se o resultado for um, é executada a décima instrução, que desativa os EPs que tiveram o resultado do último teste igual a zero e armazena o valor um no *bit* menos

significativo do resultado (R2). A décima primeira e décima segunda instruções controlam o fim do *loop*, decrementando o contador R1 e testando se ele é diferente de zero. Ao final do loop, o registrador R2 conterá o valor do maior elemento da matriz.

6. Conclusões

Os exemplos apresentados acima mostram que arquiteturas SIMD podem ter algoritmos bem simples e facilmente portáveis de máquinas seqüenciais, como no caso da soma, ou terem uma nova abordagem para melhor explorar características deste tipo de arquiteturas, como no caso do cálculo do máximo. Neste sentido, o ambiente apresentado facilita o desenvolvimento de novos algoritmos, com uma depuração bem mais fácil do que num protótipo, por exemplo. No simulador, pode-se facilmente investigar o valor de barramentos e registradores de vários EPs e da UC, simultaneamente, o que seria uma tarefa bastante difícil num protótipo.

Atualmente o projeto se encontra numa fase de desenvolvimento de rotinas básicas, tendo por objetivo criar uma biblioteca de funções paralelas envolvendo matrizes e imagens. O próximo passo é partir para o desenvolvimento de uma linguagem paralela de alto nível, com alto nível de abstração da arquitetura interna da máquina.

Referências

[Bat80] K.E.Batcher. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, C-29(9):836-840, September 1980.

[Bla90] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of the 35th IEEE Computer Society International Conference-Spring COMPCON 90*, pp. 20-24, San Francisco, CA, February 1990. IEEE Computer Society.

[Hil85] W.D.Hillis. *The Connection Machine*. The MIT Press, Massachussets, 1985.

[Jes87] C.R.Jesshope, A.J.Rushton, A.J.de O.Cruz, and J.M.Stewart. The Structure and Applications of RPA - a highly parallel adaptive architecture. In G. L. Refins and M. H. Barton, editors, *Highly Parallel Computers*, pp. 81-95. Elsevier Science Publishers, Amsterdam, 1987.

[Red73] S.F.Reddaway. The DAP Approach. In C.R.Jesshope and R.W.Hockney, editors, *Infotech State of the Art Report: Super-Computers*, pp. 311-329. Infotech Intl Ltd, Maidehead, 1973.

[Rox93] M.F.E.B.Roxo, A.J.O.Cruz and O.C.M.B.Duarte. PRIMATA: Desenvolvimento de um Elemento Processador para uma Arquitetura Massivamente Paralela. In *Anais do V Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pp. 150-165, Setembro de 1993.

[Rox94] M.F.E.B.Roxo, A.J.O.Cruz, O.C.M.B.Duarte and R.C.B.Jorge. PRIMATA: Uma Arquitetura Maciçamente Paralela para Processamento de Imagens e Matrizes. In *Anais do VI Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pp. 185-197, Agosto de 1994.