# PRIMATA: Desenvolvimento de um Elemento Processador para uma Arquitetura Massivamente Paralela

Maria Fernanda Eppinghaus Belford Roxo[1]

Adriano Joaquim de Oliveira Cruz[2]

Otto Carlos Muniz Bandeira Duarte[3]

## Resumo

Este trabalho apresenta as pesquisas realizadas para o desenvolvimento do Elemento Processador (EP) do projeto PRIMATA, que é uma arquitetura SIMD massivamente paralela orientada para o processamento de imagens e matrizes. O PRIMATA é composto por uma matriz de Elementos Processadores, controlados por uma única Unidade de Controle. Cada EP pode se comunicar com seus oito vizinhos ao N, NE, L, SE, S, SO, O e NO. A definição do Elemento Processador procurou otimizá-lo para a execução de operações aritméticas, porém sempre respeitando a restrição do custo individual de cada EP. Para isto, foram analisadas algumas propostas de arquiteturas e algoritmos para execuções das quatro operações aritméticas básicas sobre inteiros: soma, subtração, multiplicação e divisão. O resultado desta análise determinou um Elemento Processador de 4 *bits*.

## Abstract

This work presents the research done during the development of a Processing Element (PE) for the PRIMATA project, a SIMD massively parallel architecture designed for image and matrix processing. PRIMATA is an array of PEs controlled by a single Control Unit. Each PE is able to communicate with its eight nearest neighbours: N, NE, E, SE, S, SW, W, NW. The definition of the Processing Element seek to optimize the execution of arithmetic operations under the restriction of PE cost. The proposed architectures and integer arithmetic algorithms are discussed. The chosen 4 *bit* Processing Element is presented.

---

[1]Mestranda COPPE/UFRJ; Pesquisadora NCE/UFRJ; E-mail: fernanda@barra.nce.ufrj.br, Endereço: NCE/UFRJ, Cx.Postal 2324, Rio de Janeiro, RJ, CEP 20001-860

[2]Ph.D.; Pesquisador NCE/UFRJ; E-mail: adriano@barra.nce.ufrj.br

[3]Dr. Eng., Professor da COPPE/UFRJ, E-mail: duarte@masi.ibp.fr

# 1. INTRODUÇÃO

Em algumas áreas, como por exemplo, processamento de imagens, meteorologia e compressão de dados em tempo real; a capacidade de processamento demandada é bem maior do que os sistemas convencionais atuais são capazes de prover. Estas áreas têm em comum o fato de tratarem de problemas envolvendo grande quantidade de dados estruturados em forma de matrizes ou vetores. O projeto PRIMATA tem por objetivo desenvolver um computador paralelo de alto desempenho e baixo custo especialmente voltado para este tipo de aplicação. Para tal, é utilizada uma arquitetura SIMD (*Single Instruction Stream Multiple Data Stream*).

Um sistema SIMD é composto por um conjunto de elementos processadores (EPs), geralmente organizados em forma de matriz, atuando sob o controle de uma única Unidade de Controle (UC). Num sistema deste tipo todos os EPs ativos executam a mesma instrução ao mesmo tempo, porém sobre dados diferentes. A opção por uma arquitetura SIMD se deve não só às aplicações em vista, como também ao seu baixo custo e à facilidade de migração dos programas existentes. Por se tratarem de dispositivos simples, basicamente uma unidade lógica e aritmética, desprovidos de mecanismos de busca de instrução ou lógica de decodificação, os EPs têm um custo individual baixo. O desempenho da arquitetura depende fundamentalmente do produto do número de EPs pela velocidade de cada EP. Sendo assim, o desempenho individual não é crítico. O essencial é manter um compromisso entre o custo e o desempenho do EP, a fim de viabilizar o fator número de EPs. Quanto à programação, o fluxo único de instruções de um sistema SIMD permite o uso de técnicas similares às de um computador seqüencial. Isto facilita não só a adição de construções paralelas às linguagens seqüenciais convencionais, como também a migração dos programas já existentes e dos usuários destas linguagens.

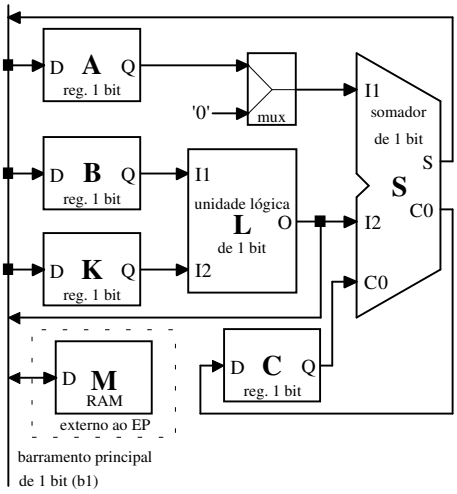


Figura 1. EP básico de 1 *bit* (EP1)

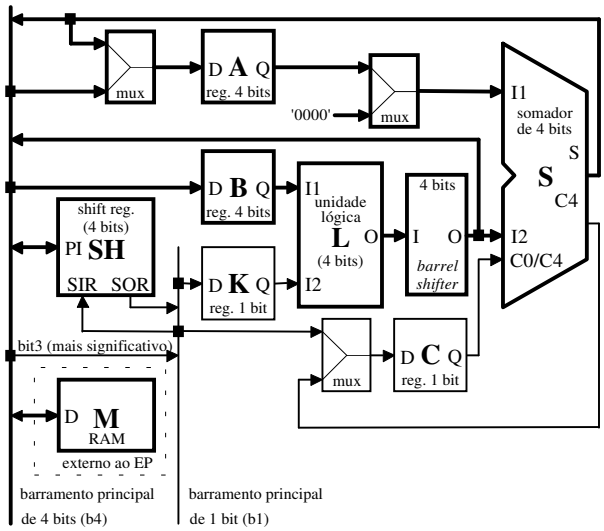


Figura 2. EP básico de 4 *bits* (EP4)

Este trabalho apresenta o processo de definição do Elemento Processador (EP) do PRIMATA, que procurou otimizar a arquitetura para a execução das operações aritméticas mais importantes, tendo como restrições o tamanho e o custo da área de silício e o número de processadores que poderemos construir. Outros fatores levados em conta foram o desejo que o projeto possa ser reproduzível no Brasil a baixo custo e que a arquitetura possa, após a sua definição, passar por um período de avaliação. Por estas razões, optamos por utilizar dispositivos lógicos programáveis (EPLDs). Estes dispositivos limitaram a largura do processador, mas esta restrição não é forte, já que arquiteturas comerciais como o DAP [Red73], o MPP [Bat80] e a CM-1 [Hil85] utilizam EPs de 1

*bit*. Para as aplicações em vista podemos trocar largura de palavras por número de processadores. A metodologia utilizada para a otimização foi a análise das operações aritméticas representativas em ordem crescente de complexidade. As figuras 1, 2, 3 e 4 apresentam as diversas propostas de arquitetura que analisaremos. Os elementos que compõem cada arquitetura serão apresentados no decorrer do trabalho, na medida em que o seu uso for se fazendo necessário.

Da mesma forma que a grande maioria dos computadores atuais, o PRIMATA tem seus números inteiros representados em complemento a dois. Nas próximas seções são apresentados algoritmos para implementação das quatro operações aritméticas básicas para inteiros: soma, subtração, multiplicação e divisão. A partir deste estudo, é determinada a melhor proposta de arquitetura para o EP.

## 2. NOTAÇÕES

Notações utilizadas nos algoritmos apresentados neste trabalho:

- Os *bits* (ou palavras) de um número inteiro são numerados do menos significativo para o mais significativo.
- As palavras de 1 *bit* são representada por letras minúsculas e as palavras de 4 *bits* são representadas por letras maiúsculas. Logo, para um número inteiro $v$, $V_i$ representa a palavra $v_{4i+3..4i}$.
- A soma $z = x + y$, onde o *carry* é denominado $c$, é representada por $(c, z) = x + y$.

## 3. SOMA

Considere a soma $z = x + y$, onde $x$, $y$ e $z$ são números inteiros representados em complemento a dois com $n$ *bits*. Para executar esta operação, basta somar os $n$ *bits* dos dois operandos e colocar o resultado em $z$, como se fosse uma soma entre inteiros sem sinal. A única diferença encontra-se no teste de *overflow*, que tem que levar em consideração os sinais dos operandos.

Resta analisar qual é a arquitetura mais adequada à realização desta operação. Para realizar uma soma, um EP simples deve conter, pelo menos, três registradores de 1 *bit* (dois para armazenar os operandos e um para armazenar o *carry*) e um somador completo de 1 *bit*. Além disso, a cada EP deve estar associado um dispositivo de memória (uma RAM com largura de 1 *bit*, por exemplo), onde serão armazenados os operandos e o resultado. Partindo destas necessidades, surge então a arquitetura EP1, apresentada na figura 1. Ela é composta pelos registradores A, B e C, pelo somador completo S e está associada à memória M. Para esta arquitetura, o algoritmo que resolve a soma é:

$C = 0$
para $i$ variando de 0 até $n - 1$
    $A = x_i$
    $B = y_i$
    $(C, z_i) = A + B + C$

Número de ciclos necessários para executar a soma: $3n + 1$.

Para melhorar a performance do EP na execução da soma, dois caminhos podem ser seguidos. O primeiro é tentar diminuir o número de iterações do *loop*, através do aumento da largura dos dados do EP. O segundo é tentar diminuir o número de iterações de cada *loop*, através de uma otimização dos acessos à memória. Pelo primeiro caminho, pode-se construir uma arquitetura idêntica a primeira, porém utilizando um EP com registradores, somador e memória com largura maior. No caso, com os EPLDs disponíveis no laboratório, podemos ir até 4 *bits*.

Surge então a arquitetura EP4 apresentada na figura 2. Para esta arquitetura, o algoritmo que resolve a soma é:

$C = 0$

para $i$ variando de 0 até $n/4 - 1$

$A = X_i$

$B = Y_i$

$(C, Z_i) = A + B + C$

Número de ciclos necessários para executar a soma: $3n/4 + 1$.

Por esta solução, o custo do processador é quadruplicado. Porém, o número de iterações do *loop* também é dividido por quatro, diminuindo cerca de quatro vezes o tempo necessário para a execução de uma soma.

A otimização dos acessos aos dados também pode ser feita através da implementação de uma memória interna ao EP, do tipo *double port* ou *triple port*. Esta solução diminui cerca de três vezes (*triple port*) o tempo de execução da soma, porém a um custo bem mais elevado do que a anterior (EP4), devido às memórias *triple port*. Logo, o aumento da largura dos dados do EP é mais barato e mais eficiente do que a implementação de uma memória interna de acesso múltiplo.

## 4. SUBTRAÇÃO

Considere a subtração $z = x - y$, onde $x$, $y$ e $z$ são números inteiros representados em complemento a dois com $n$ *bits*. Esta subtração pode ser analisada como a soma $z = x + (-y)$. Mas, $-y = \overline{y} + 1$. Logo, $z = x + \overline{y} + 1$, o que reduz o problema da subtração a uma soma (com *carry* inicialmente igual a 1) e uma complementação. Para executar estas duas tarefas, soma e complementação, simultaneamente, basta acrescentar às arquiteturas EP1 e EP4 uma unidade lógica (L).

Com esta modificação, o algoritmo que resolve a subtração para o EP1 é:

$C = 1$

para $i$ variando de 0 até $n - 1$

$A = x_i$

$B = y_i$

$(C, z_i) = A + \overline{B} + C$

Número de ciclos necessários para executar a subtração: $3n + 1$.

As soluções para melhorar a performance do EP na execução da subtração são as mesmas apresentadas para a soma. A conclusão também é a mesma: o aumento da largura dos dados do EP é mais barato e mais eficiente do que a implementação de uma memória interna de acesso múltiplo.

Sendo assim, para o EP4, após o acréscimo da unidade lógica, o algoritmo que resolve a subtração é:

$C = 1$

para $i$ variando de 0 até $n/4 - 1$

$A = X_i$

$B = Y_i$

$(C, Z_i) = A + \overline{B} + C$

Número de ciclos necessários para executar a subtração: $3n/4 + 1$.

Assim como na soma, o custo do processador é quadruplicado. Porém, o número de iterações do *loop* também é dividido por quatro, diminuindo cerca de quatro vezes o tempo necessário para a execução de uma subtração.

# 5. MULTIPLICAÇÃO

## 5.1 - Multiplicação sem Sinal

Para encontrar a melhor maneira de executar uma multiplicação entre dois números inteiros representados em complemento a dois, primeiramente, usamos o algoritmo básico de multiplicação de dois números inteiros positivos. Em seguida, foram analisadas, separadamente, as diversas combinações de sinais entre os operandos, visando chegar a um algoritmo único que englobe todos os casos.

Seja a multiplicação $p = x \times y$, onde $p$, $x$ e $y$ são números inteiros e positivos, sendo que $x$ e $y$ são representados por *n bits* e $p$ é representado por 2*n bits*. O algoritmo básico que executa a multiplicação é:

$p = 0$
para $i$ variando de 0 até $n-1$
    se $y_i = 1$, então $p = p + 2^i x$

Deve-se notar que, apesar de $p$ ser representado por 2*n bits*, as somas só precisam ter *n bits*. Isto se explica porque $2^i x$ só possui *n bits* significativos e $p$ possui os $n-i$ *bits* mais significativos iguais a zero. Desta forma, a expressão $p = p + 2^i x$ pode ser substituída por $p_{n+i \cdots i} = p_{n+i-1 \cdots i} + x_{n-1 \cdots 0}$. Sendo assim, para executar este algoritmo são necessárias $n$ comparações de 1 *bit*, uma atribuição de 2*n bits* e $n$ somas de *n bits* com armazenamento do *carry*.

O aparecimento de uma estrutura do tipo SE ... ENTÃO ... em um algoritmo leva ao surgimento do registrador de máscara (K). Este registrador é uma das características das arquiteturas SIMD. Nestas arquiteturas, todos os processadores executam a mesma instrução, o que dificulta a implementação de uma estrutura deste tipo. Através do registrador de máscara é possível modificar uma operação, tornando-a condicional. Para isto, a condição é armazenada em K. Através da unidade lógica é possível, por exemplo, executar A = A + B nos processadores em que K é igual a um e A = A nos processadores em que K é igual a zero, bastando para isto executar A = A + (B and K). A construção que reflete esta operação é ONDE <expressão booleana> FAÇA <expressão1> NO_RESTANTE <expressão2>. Todas as arquiteturas propostas neste trabalho possuem um registrador de máscara (K) ligado em uma das entradas da unidade lógica (L).

## 5.2. Combinação de sinais na multiplicação

Considerando agora a multiplicação com sinal $p = x \times y$, pode-se dividir a análise em quatro casos:

a) $x$ e $y$ são positivos ou nulos.

Multiplicando-se $x$ por $y$ sem nenhuma alteração, como se fosse uma multiplicação sem sinal, tem-se que:

$$p = |x| \times |y| = |x \times y| \equiv x \times y.$$

Logo, a multiplicação sem sinal resulta numa multiplicação correta, como era de se esperar.

b) $x$ é negativo e $y$ é positivo ou nulo.

Multiplicando-se $x$ por $y$ sem nenhuma alteração, como se fosse uma multiplicação sem sinal, tem-se que:

$$p = (2^n - |x|) \times |y| = 2^n |y| - |x \times y| \neq x \times y.$$

A multiplicação sem sinal resulta em um erro no resultado da multiplicação. Este resultado só estaria correto se $p$ fosse representado por apenas *n bits*. Existem duas formas de corrigir este erro:

- "extensão" do sinal de *x*. Por este método *x* passa a ser representado por 2*n bits*, resultando em:

$$p = (2^{2n} - |x|) \times |y| = 2^{2n}|y| - |x \times y| = -|x \times y| \equiv x \times y$$

- subtração do termo em excesso, isto é, $2^n\ y = 2^n |y|$. Por este método, a multiplicação fica:

$$p = (2^n - |x|) \times |y| - 2^n |y| = 2^n |y| - |x \times y| - 2^n |y| = -|x \times y| \equiv x \times y$$

c) *x* é positivo ou nulo e *y* é negativo.

Multiplicando-se *x* por *y* sem nenhuma alteração, como se fosse uma multiplicação sem sinal, tem-se que:

$$p = |x| \times (2^n - |y|) = 2^n |x| - |x \times y| \neq x \times y .$$

A multiplicação sem sinal resulta em um erro no resultado da multiplicação. Este resultado só estaria correto se *p* fosse representado por apenas *n bits*. Para corrigir este erro, pode-se usar um dos dois métodos sugeridos para *x* no caso anterior, como é mostrado a seguir:

- "extensão" do sinal de *y* para 2*n bits*. Por este método a multiplicação fica:

$$p = |x| \times (2^{2n} - |y|) = 2^{2n}|x| - |x \times y| = -|x \times y| \equiv x \times y .$$

- subtração do termo em excesso, isto é, $2^n\ x = 2^n |x|$. Por este método, a multiplicação fica:

$$p = |x| \times (2^n - |y|) - 2^n |x| = 2^n |x| - |x \times y| - 2^n |x| = -|x \times y| \equiv x \times y$$

Neste caso, a subtração do termo em excesso é bem mais vantajosa. A "extensão" do sinal de *y* dobraria o número de somas do algoritmo e, conseqüentemente, o tempo necessário para executar a multiplicação.

d) *x* e *y* são negativos.

Multiplicando-se *x* por *y* sem nenhuma alteração, como se fosse uma multiplicação sem sinal, tem-se que:

$$p = (2^n - |x|) \times (2^n - |y|) = 2^{2n} - 2^n (|x| + |y|) + |x \times y| \neq x \times y .$$

A multiplicação sem sinal resulta em um erro no resultado da multiplicação. Este resultado só estaria correto se *p* fosse representado com apenas *n bits*. Combinando as modificações propostas nos dois casos anteriores, chega-se a duas formas de correção:

- "extensão" do sinal de *x* e subtração do termo em excesso em *x*, que leva a seguinte expressão:

$$p = (2^{2n} - |x|) \times (2^n - |y|) - 2^n (2^{2n} - |x|) = |x \times y| - 2^{2n}|y| = |x \times y| \equiv x \times y$$

- subtração dos dois termos em excesso, levando a:

$$p = (2^n - |x|) \times (2^n - |y|) - 2^n (2^n - |y|) - 2^n (2^n - |x|) = |x \times y| - 2^{2n} = |x \times y| \equiv x \times y$$

| | $x \geq 0$ | $x < 0$ |
|---|---|---|
| $y \geq 0$ | $p = x \times y$ | $p = x' \times y$ ou $p = x \times y - 2^n y$ |
| $y < 0$ | $p = x \times y - 2^n x$ | $p = x' \times y - 2^n x$ ou $p = x \times y - 2^n y - 2^n x$ |

Tabela 1 - Correções para a multiplicação de inteiros com sinal

A tabela 1 apresenta um resumo das formas de executar a multiplicação propostas para cada um dos quatro casos analisados. Para esta tabela, considere que $x \times y$ é a multiplicação de *x* por *y* sem levar em consideração os sinais dos fatores e $x'$ é a extensão do sinal de *x* de *n* para 2*n bits*. Analisando a tabela 1 é possível deduzir dois métodos gerais para a multiplicação.

**1º Método** - Extensão do sinal de $x$ e subtração do termo em excesso em $x$. Considere $x'$ a extensão de sinal de $x$ de $n$ para $2n$ *bits*. O algoritmo básico para a implementação deste método é:

$p = 0$
para $i$ variando de 0 até $n-1$
    se $y_i = 1$, então $p = p + 2^i x'$
se $y < 0 (y_{n-1} = 1)$, então $p = p - 2^n x'$

Este algoritmo pode ser bastante melhorado. Inicialmente, o primeiro passo (atribuição do valor inicial zero ao resultado da operação) e a primeira iteração de soma ($i = 0$) podem ser condensados em um único passo. Ao invés de somar $x'$ a $p$ se $y_0 = 1$, pode-se atribuir diretamente o valor de $x'$ a $p$ (se $y_0 = 1$) ou atribuir 0 (zero) a $p$ (se $y_0 = 0$). Além disso, a última iteração de soma e a subtração podem ser condensadas em uma única operação de subtração. Ambas têm o mesmo fator condicionante, ou seja, $y_{n-1} = 1$. Neste caso, serão executadas as seguintes operações: $p = p + 2^{n-1} x'$ e $p = p - 2^n x'$. Juntando as duas operações tem-se: $p = p + 2^{n-1} x' - 2^n x' = p - 2^{n-1} x'$.

Quanto às operações condicionais do tipo se ..., então ... senão ..., elas são implementadas através do registrador condicional K. Por último, as somas e a subtração não precisam ser de $2n$ *bits* cada uma, o que melhora o desempenho do algoritmo para máquinas que trabalham com operandos de $n$ *bits* ou menos. Na verdade, são necessárias apenas somas de $n$ *bits* acrescidas de uma extensão de sinal de 1 *bit*.

Após as modificações propostas, o algoritmo de multiplicação fica:

$K = y_0$
$p_{n..0} = x'_{n..0}$ and $K$
para $i$ variando de 1 até $n-2$
    $p_{n+i} = p_{n+i-1}$
    $K = y_i$
    $p_{n+i..i} = p_{n+i..i} + x'_{n..0}$ and $K$
$p_{2n-1} = p_{2n-2}$
$K = y_{n-1}$
$p_{2n-1..n-1} = p_{2n-1..n-1} - x'_{n..0}$ and $K$

Para executar este algoritmo são necessárias: uma atribuição de $n+1$ *bits*, $2n-1$ atribuições de 1 *bit*, $n-2$ somas de $n+1$ *bits* e uma subtração de $n+1$ *bits*.

**2º Método** - Subtração dos termos em excesso em $x$ e $y$. O algoritmo básico para a implementação deste método é:

$p = 0$
para $i$ variando de 0 até $n-1$
    se $y_i = 1$, então $p = p + 2^i x$
se $y < 0 (y_{n-1} = 1)$, então $p = p - 2^n x$
se $x < 0$ $(x_{n-1} = 1)$, então $p = p - 2^n y$

Este algoritmo também pode ser bastante melhorado. Assim como no algoritmo do 1º método, o primeiro passo e a primeira iteração de soma podem ser condensados em uma única operação. Da mesma forma, a última iteração de soma e a subtração do termo em excesso em $x$ também podem ser condensadas em uma única

operação. Além disso, as operações condicionais são implementadas através do registrador condicional K. Com as modificações propostas, o algoritmo de multiplicação fica:

$K = y_0$

$p_{n-1..0} = x_{n-1..0}$ and $K$

$p_n = 0$

para $i$ variando de 1 até $n-2$

$K = y_i$

$p_{n+i..i} = p_{n+i-1..i} + x_{n-1..0}$ and $K$

$K = y_{n-1}$

$p_{2n-1..n-1} = p_{2n-2..n-1} - x_{n-1..0}$ and $K$

$K = x_{n-1}$

$p_{2n-1..n} = p_{2n-1..n} - y_{n-1..0}$ and $K$

Para executar este algoritmo são necessárias: $n+1$ comparações de 1 *bit*, uma atribuição de $n+1$ *bits*, $n-2$ somas de *n bits* com armazenamento do *carry* e duas subtrações de *n bits*, sendo que uma com armazenamento do *carry*.

Dependendo das características da arquitetura, um ou outro método se mostra mais eficiente. De forma geral, o 1º método é mais adequado aos processadores de 1 *bit*, enquanto que o 2º método é mais adequado aos processadores que lidam com palavras maiores.

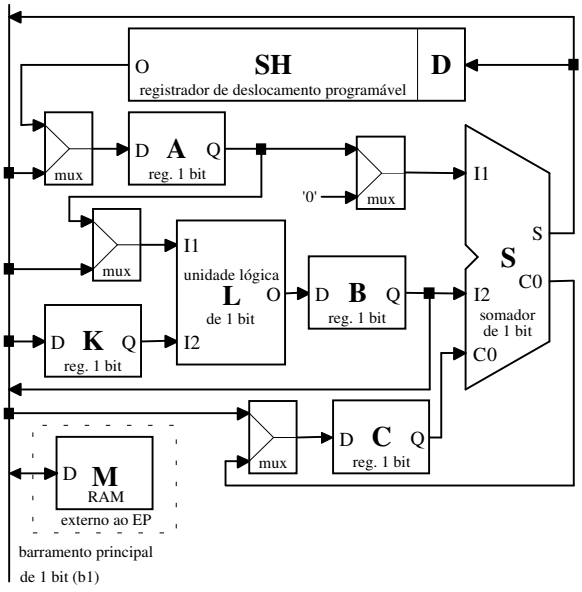


Figura 3. EP de 1 *bit* com registrador de deslocamento programável (EP1SH)

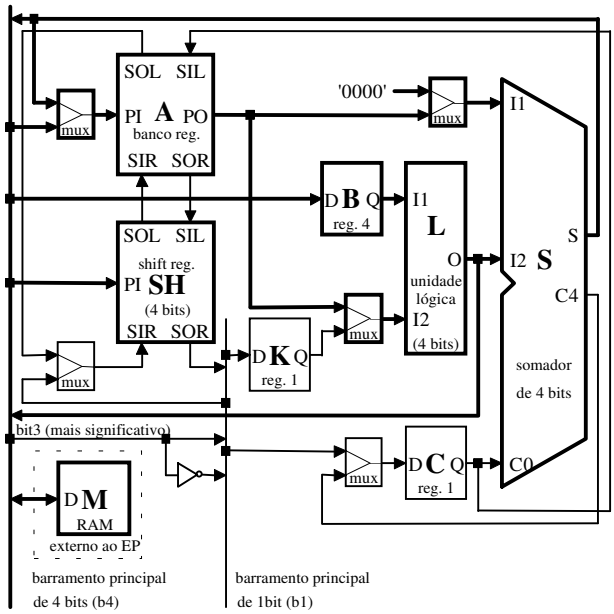


Figura 4. EP de 4 *bits* com banco de registradores (EP4BR)

Para implementar a multiplicação no EP1 (figura 1), é utilizado o algoritmo do 1º método. O custo da multiplicação para o programa gerado é de $3n^2 + n$ ciclos. O fator 3 (três) advém do acesso à memória. Para cada soma (ou subtração) de 1 *bit*, são lidos da memória 1 *bit* de *p* e 1 *bit* de *x* e o resultado (1 *bit* de p) é escrito na memória. Logo, cada soma (ou subtração) necessita de $3n+1$ ciclos para a sua execução.

É fácil perceber que o gargalo do sistema é o acesso à memória. A solução encontrada para melhorar o desempenho da multiplicação está na modificação do *hardware*, através da criação de uma memória interna de

acesso imediato, que permita o armazenamento dos resultados parciais de *p*, economizando dois acessos à memória externa para cada operação sobre um *bit*. Surge então a arquitetura EP1SH, mostrada na figura 3. Ela é baseada no EP1, porém implementa uma memória interna através do seu registrador de deslocamento de tamanho programável (SH), utilizado para o armazenamento de resultados parciais. Este registrador tem o *bit* mais significativo denominado D e o menos significativo ligado a entrada do registrador A. Esta solução se assemelha a apresentada pelo MPP. Para executar a multiplicação o registrador de deslocamento deve ser configurado com tamanho $n-1$. Durante o laço principal, é possível armazenar os resultados parciais que serão utilizados na próxima soma no conjunto: registrador de deslocamento SH, registrador A. Este conjunto tem um total de *n bits* de largura. Desta forma, adaptando o algoritmo do 1º método ao EP1SH, obtêm-se um desempenho bem melhor do que o obtido para o EP1. A multiplicação é processada em $n^2+3n$ ciclos.

Para analisar os processadores de 4 *bits*, será considerado que o tamanho dos operandos da multiplicação é um múltiplo de quatro. Na prática, esta não é uma restrição na maior parte dos casos. Considere um processador que trabalha com palavras de 4 *bits*. A fim de explorar ao máximo as potencialidades deste tipo de processador, deve-se procurar buscar utilizar, sempre que possível, operações que manipulem palavras de 4 *bits*. No algoritmo do 1º método, todas as operações atuam sobre dados de 1 *bit* ou de $n+1$ *bits*. Desta forma, este método não explora bem o processador. Por outro lado, o 2º método executa uma série de operações sobre *n bits* (que é um múltiplo de quatro), explorando melhor e portanto se adaptando melhor aos processadores de 4 *bits*.

No caso dos processadores de 4 *bits*, o algoritmo da multiplicação apresenta um problema para a sua implementação. Os dados são lidos e escritos na memória de 4 em 4 *bits*, já que tanto a memória quanto os registradores têm largura de 4 *bits*. Porém, tanto a leitura quanto a escrita dos resultados parciais nem sempre são executadas sobre o mesmo conjunto de 4 *bits*. Por exemplo, na atribuição inicial, *p* é dividido em palavras de 4 *bits* a partir do *bit* 0. Já na primeira soma, *p* é dividido em palavras de 4 *bits* a partir do *bit* 1. Isto ocorre porque a multiplicação é composta por um conjunto de somas com deslocamento de apenas um *bit*. Sendo assim, a primeira soma (substituída pela atribuição inicial) começa a partir do *bit* 0, a segunda soma começa a partir do *bit* 1 e assim por diante. Implementar uma memória em que se possa ter acesso a qualquer subconjunto de 4 *bits* não é uma tarefa muito simples, muito menos de baixo custo. Logo, faz-se necessário encontrar outra solução. A solução mais obvia é a utilização de um *barrel shifter* para deslocar um dos operandos, quantas posições sejam necessárias. Foi então acrescentado ao EP4 um *barrel shifter* entre a saída da unidade lógica (L) e a entrada do somador (S), capaz de deslocar um operando de 4 *bits* de até quatro posições para a esquerda ou para a direita.

O custo desta solução em termos de desempenho é o aumento do número de somas. Quando uma soma necessita de algum deslocamento, ela é subdividida em subsomas. Deve-se tomar especial cuidado com a propagação do *carry* nestas subsomas. Para melhor implementar estas subsomas, foi feita uma pequena alteração no somador do EP4. O somador proposto tem duas entradas de 4 *bits* (para os operandos) e duas entradas de *carry* (uma no *bit* 0 e outra no *bit* 4). Para facilitar a implementação, apenas uma delas pode ser utilizada de cada vez, ou seja, um *bit* de controle indica qual a posição do *carry*. Com esta modificação no somador foi possível implementar somas de *n bits* com deslocamento em $3n/4+2$ ciclos, ou seja, apenas um ciclo a mais do que uma soma sem deslocamento (seção 3).

Outro problema encontrado no desenvolvimento da multiplicação para o EP4 foi a inexistência de uma forma de comunicação de dados entre os dispositivos de 1 e 4 *bits*. Para resolver este problema foi acrescentado o

registrador de deslocamento de 4 bits (SH), cuja entrada/saída paralela está ligada ao barramento principal de 4 *bits* e a entrada/saída serial está ligada ao barramento principal de 1 *bit*. Além disto, a fim de aliviar o fluxo de dados no barramento principal de 4 *bits*, foi acrescentada uma ligação direta entre a saída do somador e a entrada do registrador A. Com estas modificações, aplicando o algoritmo de multiplicação do 2º método ao EP4, chega-se a um programa que executa a multiplicação em $12(n/4)^2 + 10n/4 + 1$ ciclos. A maior parte do tempo de execução é consumido pelas somas que, além do problema do deslocamento de *bits*, sofrem com a falta de memória interna.

O desempenho do EP4 para a multiplicação ficou aquém do desejável, considerando-se o custo associado a um processador de 4 *bits*. O superioridade desta arquitetura em relação ao EP1SH é muito pequena. A solução encontrada para resolver os problemas de falta de memória interna e dificuldade de operar dados deslocados é a substituição do registrador A por um banco de registradores A, contendo 8 registradores de deslocamento de 4 *bits* conectados entre si, formando um grande registrador de deslocamento de 32 *bits*. Somente as operações de deslocamento são realizadas em todos os registradores. Tanto as operações de leitura quanto as operações de escrita são realizadas em um único registrador de deslocamento de 4 *bits* de cada vez. Para selecionar qual dos 8 registradores será utilizado para a leitura ou escrita são utilizados 3 *bits* de controle. Surge então o EP4BR, mostrado na figura 4. Ele é baseado no EP4, porém substitui o registrador A pelo banco de registradores A descrito acima, tornando dispensáveis o *barrel shifter* e a entrada de *carry* multiplexada do somador. Na implementação da multiplicação para o EP4BR é utilizado o algoritmo do 2º método, associado ao armazenamento de resultados parciais no banco de registradores A. Foram desenvolvidos dois programas de multiplicação: um para fatores de até 28 *bits* e outro para fatores de 32 *bits*. Esta separação é uma decorrência da escolha da capacidade de armazenamento do banco de registradores, que é de apenas 32 *bits*. Para multiplicações com fatores de 32 *bits* são necessários até 36 *bits* para armazenamento de resultados parciais. Para contornar este problema, nestas multiplicações, o registrador de deslocamento SH é utilizado como dispositivo de armazenamento auxiliar dos resultados parciais. Para isto, uma comunicação serial entre o banco de registradores (A) e o registrador de deslocamento (SH) foi criada. O programa que executa a multiplicação para fatores de até 28 *bits* demanda $4(n/4)^2 + 7n/4 + 2$ ciclos e o para fatores de 32 *bits* demanda $4(n/4)^2 + 9n/4$ ciclos.

O desempenho deste EP4BR na execução da multiplicação é claramente superior ao desempenho de todas as outras propostas de arquiteturas apresentadas. Além disto, o seu custo não é excessivamente elevado: o seu número de registradores é ligeiramente maior do que o do EP1SH e a complexidade da sua lógica é comparável a do EP4.

# 6. DIVISÃO

## 6.1. Divisão Com Restauração e Sem Restauração

Assim como na multiplicação, a análise da divisão será separada em duas partes: divisão de inteiros positivos e divisão de inteiros com sinal.

Considere a divisão de $x$ por $y$, tendo como resultados o quociente $q$ e o resto $r$, onde $x$, $y$, $q$ e $r$ são números inteiros positivos representados por $n$ *bits*. Pode-se afirmar que:

$$x = q \times y + r, \text{ onde } 0 \le r < y$$

Um algoritmo simples para executar a divisão de $x$ por $y$ é:

$x'$ = expansão de $x$ para $2n-1$ bits
para $i$ variando de 1 até $n$
    $x' = x' - 2^{n-i}\, y$
    se $x' \geq 0$, então
        $q_{n-i} = 1$
    senão
        $q_{n-i} = 0$
        $x' = x' + 2^{n-i}\, y$
$r = x'_{n-1..0}$

Este algoritmo é denominado divisão com restauração (*restoring division*). Para fazer a comparação, $x'$ é subtraído de $2^{n-i}\, y$. Esta subtração adianta a operação que tem que ser executada caso a comparação tenha resultado positivo. Por outro lado, se a comparação tiver resultado negativo, a subtração não deve ser executada. Neste caso, o valor de $x'$ tem que ser restaurado através de uma soma. Daí vem o nome divisão com restauração. Pode-se eliminar as operações de soma através de uma compensação na próxima subtração. Considere que, para $i = k$, $x' - 2^{n-i}\, y < 0$, ou seja, $x' - 2^{n-k}\, y < 0$. A restauração é feita através da operação $x' = x' + 2^{n-k}\, y$: Na iteração seguinte ($i = k+1$) é executada a operação $x' = x' - 2^{n-i}\, y$, ou seja, $x' = x' - 2^{n-k-1}\, y$. Juntando as duas operações tem-se: $x' = x' + 2^{n-k}\, y - 2^{n-k-1}\, y = x' + 2^{n-k-1}\, y(2-1) = x' + 2^{n-k-1}\, y$. Logo, a restauração pode ser feita simplesmente invertendo o sinal da próxima subtração, gerando uma soma. Utilizando-se este artifício, pode-se modificar o algoritmo para:

$x'$ = expansão de $x$ para $2n-1$ bits
*sinal* $= +1$
para $i$ variando de 1 até $n$
    $x' = x' - sinal.2^{n-i}\, y$
    se $x' \geq 0$, então
        $q_{n-i} = 1$
        *sinal* $= +1$
    senão
        $q_{n-i} = 0$
        *sinal* $= -1$
se *sinal* $= +1$, então $r = x'_{n-1..0}$
senão $r = x'_{n-1..0} + y$

Este algoritmo é denominado divisão sem restauração (*non-restoring division*). Ele é claramente superior ao algoritmo de divisão com restauração.

## 6.2. Divisão com Sinal

A primeira diferença entre as divisões de inteiros positivos e com sinal está no tamanho do resultado. Considere a divisão de $x$ por $y$, tendo como resultados o quociente $q$ e o resto $r$, onde $x$, $y$, $q$ e $r$ são números inteiros representados em complemento a dois. Considere ainda que $x$ e $y$ são representados por *n bits*. O resto $r$ é menor do que $y$, logo ele também pode ser representado por *n bits*. A diferença se encontra no quociente. O maior valor absoluto para o quociente ocorre para a divisão entre o maior valor absoluto de $x$ e o menor valor absoluto (diferente de zero) de $y$. Ou seja, o maior valor absoluto do quociente é $2^n\ (= 2^n / 1)$. Porém, este valor só pode ser representado por *n bits* se o seu sinal for negativo. Para a divisão $-2^n / -1$, o quociente é $+2^n$, o que significa

que ele precisa ser representado por pelo menos $n+1$ *bits*.

A maneira mais óbvia de resolver o problema da divisão de inteiros com sinal é calcular os módulos do dividendo e do divisor, executar uma divisão sem sinal usando os módulos e, depois, calcular os sinais do quociente e do resto. Aproveitando o algoritmo de divisão sem restauração, chega-se a:

$x' =$ expansão de $|x|$ para $2n-1$ bits
$y' = |y|$
$sinal = +1$
para $i$ variando de 1 até $n$
    $x' = x' - sinal.2^{n-i}\, y'$
    se $x' \geq 0$, então
        $q_{n-i} = 1$
        $sinal = +1$
    senão
        $q_{n-i} = 0$
        $sinal = -1$
se $sinal = +1$, então $r = x'_{n-1..0}$
senão $r = x'_{n-1..0} + y'$
se $x < 0$, então $r = -r$
se $x$ e $y$ têm sinais opostos, então $q = -q$

Para executar uma divisão através deste algoritmo são necessários: um cálculo de módulo de $2n-1$ *bits*, e outro de $n$ *bits*, $n+3$ comparações, $n+1$ somas ou subtrações de $n$ *bits*, $2n+1$ atribuições de 1 *bit* e duas inversões de sinal de $n$ *bits*.

Existem outros algoritmos para divisão com sinal, também derivados do princípio da divisão sem restauração, que dispensam o cálculo dos módulos do dividendo e/ou do divisor. Porém, eles necessitam de operações condicionais mais complexas. Isto significa que, para aproveitar todas as suas potencialidades, o EP deve dispor de mais dispositivos de mascaramento. Além disso, ao final do laço principal do algoritmo, a mesma quantidade de correções é necessária. Logo, não há vantagem no uso destes algoritmos em arquiteturas SIMD.

A implementação do algoritmo apresentado depende da arquitetura utilizada. Cada caso será analisado detalhadamente mais adiante. Porém, algumas considerações de caráter geral podem ser apresentadas desde já.

A primeira se refere ao mapeamento das variáveis auxiliares $x'$ e $y'$ na memória. Observando o algoritmo, pode-se perceber que a primeira vez em que é atribuído um valor ao resto $r$ corresponde a última vez em que é utilizada a variável $y'$. Logo, a variável auxiliar $y'$ pode ser mapeada na mesma região de memória reservada para o resto $r$. Quanto a variável $x'$, o seu mapeamento é dependente da arquitetura. Se o processador não possui memória interna, uma região da memória externa deve ser alocada para o mapeamento de variáveis auxiliares de cálculo. Este espaço deve ser reservado para o uso exclusivo do processador e o seu endereço será denominado $z$. Deve-se tomar cuidado para que não seja permitido ao usuário alocar variáveis nesta região. Por outro lado, se o processador possui memória interna, a variável $x'$ pode ser alocada nela. Nem sempre a memória interna é suficiente para alocar toda a variável $x'$. Quando isto ocorre, a solução é alocar os *bits* mais significativos de $x'$ (os primeiros a serem utilizados no algoritmo) na memória interna e utilizar a região de memória reservada para o quociente $q$ para armazenar os *bits* menos significativos. Esta solução só é possível porque, para cada *bit* do quociente $q$ calculado, pode-se dispensar um *bit* de $x'$ (o mais significativo) e, conseqüentemente, transportar um

*bit* de $x'$ da região reservada ao quociente $q$ para a memória interna.

Outra consideração importante se refere a variável *sinal*. A ela só podem ser atribuídos dois valores: +1 e -1. Logo, um único *bit* é capaz de representá-la. Resta decidir quem vai implementá-la e quais serão as representações para estes dois valores. Para implementá-la pode-se utilizar o registrador de máscara (K), presente em todas as arquiteturas apresentadas. Quanto a representação, observe que, após cada atribuição de um *bit* ao quociente $q$, uma atribuição de um *bit* é feita à variável *sinal*. Sendo assim, a fim de agilizar o algoritmo, deve-se escolher uma representação tal que o mesmo valor seja atribuído ao quociente $q$ e à variável *sinal*. Ou seja, para o algoritmo apresentado: $K = 0$ corresponde a $sinal = -1$ e $K = 1$ corresponde a $sinal = +1$.

Por último, uma análise mais cuidadosa permite concluir que o cálculo da extensão do módulo de $x$ para $2n-1$ *bits* não é necessário. Como $x$ é representado por $n$ *bits*, o seu módulo também pode ser representado por $n$ *bits*. Portanto, a extensão do módulo, ou seja, os $n-1$ *bits* restantes, serão iguais a zero. Se o resultado da extensão já é conhecido de antemão, o seu cálculo não é necessário. É mais eficiente calcular o módulo de $x$ com apenas $n$ *bits*. Na primeira iteração do laço principal do algoritmo, os $n-1$ *bits* restantes serão necessários na subtração. Neste momento, basta utilizar o valor zero como operando da subtração. Para isto a primeira iteração será separada do laço principal.

A partir das análises apresentadas acima, foram deduzidos três algoritmos básicos para a execução da divisão entre inteiros com sinal: um para processadores sem memória interna, uma para processadores com memória interna de pelo menos $n$ *bits* e um para processadores com memória interna de pelo menos $2n$ *bits*. Aqui será apresentado apenas o algoritmo para processadores com dispositivos de memória de pelo menos $2n$ *bits* (variável $x'$ mapeada na memória interna, denominada *mi*). Para executar o algoritmo apresentado abaixo são necessários: dois cálculos de módulo de $n$ *bits*, $n+1$ somas ou subtrações de $n$ *bits*, $n+3$ atribuições de 1 *bit* e duas inversões de sinal de $n$ *bits*.

$mi_{n-1..0} = |\mathrm{x}|$
$r = |y|$
$K = C = 1$
$mi_{2n-2..n-1} = (00\ldots0, mi_{n-1}) + (r_{n-1..0} \text{ xor } K) + C$
se $mi_{2n-2..n-1} \geq 0$, então $q_{n-1} = K = C = 1$
senão $q_{n-1} = K = C = 0$
para $i$ variando de 2 até $n$
    $mi_{2n-i-1..n-i} = mi_{2n-i-1..n-i} + (r_{n-1..0} \text{ xor } K) + C$
    se $mi_{2n-i-1..n-i} \geq 0$, então $q_{n-i} = K = C = 1$
    senão $q_{n-i} = K = C = 0$
$r_{n-1..0} = mi_{n-1..0} + (r_{n-1..0} \text{ and } \overline{K})$
$K = C = x_{n-1}$
$r_{n-1..0} = (r_{n-1..0} \text{ xor } K) + C$
$K = C = K \text{ xor } y_{n-1}$
$q_{n..0} = (q_{n-1..0} \text{ xor } K) + C$

Para implementar a divisão para o EP1, foi utilizado o algoritmo para processadores sem memória interna. O desempenho do programa desenvolvido está longe do ideal. Ele processa a divisão em $3n^2 + 10n + 5$ ciclos. A análise do desempenho desta arquitetura para a divisão é bem parecida com a análise da multiplicação. A

conclusão é a mesma: o gargalo do sistema é o acesso à memória. Sendo assim, a solução proposta também é a mesma, ou seja: criação de uma memória interna, como no EP1SH. Na divisão, o registrador de deslocamento de tamanho programável (SH) é utilizado para o armazenamento de $x'$. Esta arquitetura se encaixa no caso em que os *bits* mais significativos de $x'$ são alocados na memória interna e os *bits* menos significativos são armazenados na região da memória reservada para o quociente *q*. O desempenho do programa desenvolvido para este processador já é bem melhor do que o do EP básico de 1 *bit*. Ele processa a divisão em $n^2 + 10n + 5$ ciclos, ou seja, o fator três (correspondente aos acessos à memória) foi eliminado.

Alguns detalhes da arquitetura EP4 facilitam especialmente a divisão. A ligação entre o *bit* mais significativo do barramento principal de 4 *bits* e a entrada do barramento principal de 1 *bit*, por exemplo, agilizar o teste sobre o sinal do resultado das somas/subtrações do laço principal. Além disto, o registrador de deslocamento bidirecional (SH) tem um papel fundamental no armazenamento dos resultados parciais do quociente *q*. Quanto à implementação de um algoritmo de divisão para o EP4, inicialmente vale ressaltar que o algoritmo desenvolvido para a implementação de somas entre variáveis armazenadas em memórias de 4 *bits* com deslocamento não múltiplo de quatro, usado na multiplicação, também vale para as somas e subtrações da divisão.

Como o EP4 não possui dispositivo de memória interna, o algoritmo adaptado foi o mesmo utilizado para o EP1, ou seja, o algoritmo para processadores sem memória interna. O desempenho do programa desenvolvido é ainda melhor do que o do EP1SH. Ele processa a divisão em $12(n/4)^2 + 16n/4 + 7$ ciclos.

Assim como na multiplicação, o desempenho do EP4 ficou aquém do desejável, considerando-se o seu custo. Da mesma forma, a solução encontrada é a utilização de um banco de registradores, como o do EP4BR. A implementação da divisão para o EP4BR é dependente do tamanho dos operandos, devido às limitações do tamanho da memória interna. Foram desenvolvidos três programas de divisão, cada um com um limite diferente para o tamanho dos operandos. O primeiro programa armazena a variável auxiliar $x'$ inteiramente na memória interna, limitando o tamanho dos operandos a 16 *bits*. Para este programa a divisão é processada em $4(n/4)^2 + 10(n/4) + 9$ ciclos. Já o segundo e terceiro programas armazenam apenas a parte alta (mais significativa) da variável $x'$ na memória interna. A parte baixa (menos significativa) é armazenada na área da memória externa reservada para o quociente *q* e, no caso do terceiro programa, também no registrador de deslocamento (SH). O segundo programa limita o tamanho dos operandos a 28 *bits*. A utilidade deste programa é bastante limitada, visto que normalmente se trabalha com operandos que são potências de dois, podendo portanto utilizar o primeiro programa apresentado, que é mais rápido. Ele processa a divisão em $4(n/4)^2 + 14(n/4) + 4$ ciclos. Por fim, o terceiro programa é específico para a divisão de inteiros representados por 32 *bits*. Ele processa a divisão em $4(n/4)^2 + 15(n/4) + 3$ ciclos.

## 7. ESCOLHA DO ELEMENTO PROCESSADOR

A tabela 2 apresenta uma comparação entre os Elementos Processadores propostos na execução das operações de soma, subtração, multiplicação e divisão de números inteiros representados em complemento a dois. Os desempenhos são apresentados em termos de número de ciclos de máquina para a execução de uma operação sobre operandos representados por *n bits*. Para facilitar a comparação, o gráfico 1 apresenta uma comparação visual dos desempenhos das mesmas arquiteturas apenas para as operações sobre operandos de 16 *bits*.

Através das tabelas expostas, não é difícil concluir que o Elemento Processador de 4 *bits* com banco de

registradores tem o melhor desempenho, dentre as arquiteturas apresentadas, para todas as operações analisadas. É bem verdade que o seu custo também é o mais elevado, porém a superioridade do seu desempenho compensa este aumento de custo. Observe que ele possui quase o mesmo número de registradores do que os EPs de 1 *bit* com memória interna, porém ele chega a ser quase quatro vezes mais veloz do que estes processadores. Quanto a lógica necessária a um EP que lida com dados de 4 *bits*, ele chega a ser quase três vezes mais veloz do que o EP simples de 4 *bits*, que tem a mesma complexidade lógica. Sendo assim, o EP de 4 *bits* com banco de registradores foi considerado a melhor proposta de Elemento Processador para as operações aritméticas básicas sobre números inteiros.

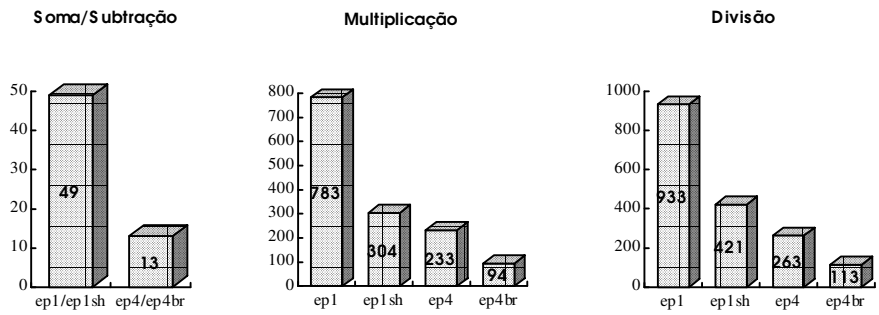


Gráfico 1 - Ciclos necessários à execução de operações sobre inteiros de 16 *bits* para diversas propostas de EPs.

| | Soma/Subtração | Multiplicação | Divisão |
|---|---|---|---|
| EP1 | $3n+1$ | $3n^2+n$ | $3n^2+10n+5$ |
| EP1SH | $3n+1$ | $n^2+3n$ | $n^2+10n+5$ |
| EP4 | $3n/4+1$ | $12(n/4)^2+10n/4+1$ | $12(n/4)^2+16n/4+7$ |
| EP4BR | $3n/4+1$ | $n \le 28$: $4(n/4)^2+7n/4+2$<br>$n=32$: $4(n/4)^2+9n/4$ | $n \le 16$: $4(n/4)^2+10n/4+9$<br>$n=32$: $4(n/4)^2+15n/4+3$ |

Tabela 2 - Ciclos necessários para a execução de operações aritméticas sobre números inteiros representados em complemento a dois por n *bits* para as diversas propostas de EPs.

Por último, a tabela 3 apresenta o desempenho de algumas arquiteturas comerciais [Red73,Bat80,JHC87,Bla90] na execução da soma, subtração e multiplicação de inteiros representados em complemento a dois por 8 *bits*. Comparando os resultados apresentados na tabela 2 com os resultados apresentados na tabela 3, pode-se concluir que o EP de 4 *bits* com banco de registradores só perde em desempenho para o MASPAR. Porém, o EP do MASPAR tem um custo bem mais elevado, possuindo mais de 1536 *bits* de memória interna, contra 42 *bits* do ep4br, além de uma lógica bem mais complexa. Deve-se considerar também que, os tempos apresentados para o MASPAR supõem que um dos operandos está num registrador especial e o resultado da operação será armazenado neste mesmo registrador especial, o que é uma restrição importante.

| Operação | DAP510 | MPP | RPA | MASPAR | EP4BR |
|---|---|---|---|---|---|
| Soma/Subtração | 28 | 25 | 11 | 2 | 7 |
| Multiplicação | 250 | 88 | 80 | 25 | 32 |

Tabela 3 - Ciclos necessários à execução de operações sobre inteiros de 8 *bits* para máquinas SIMD comerciais.

## 8. CONEXÃO ENTRE OS ELEMENTOS PROCESSADORES

O PRIMATA permite uma comunicação de 1 *bit* de largura entre cada EP e seus oito EPs vizinhos: norte, nordeste, leste, sudeste, sul, sudoeste, oeste e noroeste. Para isto é utilizado o mesmo esquema adotado pelo BLITZEN [BDR0], com cada EP associado a quatro linhas direcionadas aos vizinhos: NE, SE, SO e NO. Na interseção das linhas que se cruzam há uma conexão. Durante o roteamento, cada EP envia dados em uma das quatro direções e recebe dados de uma das direções restantes. As direções de envio e recebimento definem o fluxo dos dados.

## 9. CONSIDERAÇÕES FINAIS

Após a análise das operações aritméticas sobre números inteiros analisamos as operações sobre reais (ponto-flutuante), porém utilizando apenas o EP4BR. Criamos algoritmos de soma e multiplicação em ponto-flutuante e fizemos algumas pequenas modificações no EP em relação a versão aqui apresentada, a fim de melhorar o desempenho da arquitetura na execução das operações sobre reais.

Todos os algoritmos desenvolvidos foram testados num simulador lógico especialmente desenvolvido para este projeto. Já conseguimos integrar uma versão do EP em um único circuito integrado do tipo EPLD. Caso haja disponibilidade de *gate arrays* programáveis no campo (FPGAs), esperamos colocar quatro EPs em um circuito integrado.

## 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*, pages 20-24, San Francisco, CA, February 1990. IEEE Computer Society.

[BDR0] D. W. Blevins; E. W. Davis, and J. H. Reif. *Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor.*

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

[JHC87] C. R. Jesshope, A. J. Rushton, A. J. de O. Cruz, and J. M. Stewart. The Structure and Aplications of RPA - a highly parallel adaptive architecture. In G. L. Refins and M. H. Barton, editors, *Highly Parallel Computers*, pages 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*, pages 311-329. Infotech Intl Ltd, Maidehead, 1973.