CAPÍTULO III

OPERAÇÕES ARITMÉTICAS BÁSICAS ENVOLVENDO NÚMEROS INTEIROS

III.1 - Notações

Antes de iniciar as discussões a respeito da representação e das operações envolvendo números inteiros, vale a pena apresentar algumas notações que serão utilizadas no decorrer deste capítulo. As notações apresentadas também são válidas para o resto deste trabalho.

- Representação de um número:
  - Seja v um número representado por n *bits*.
    - os *bits* de v são numerados do menos significativo para o mais significativo. Logo:
      - o *bit* menos significativo de v é representado por v_0,
      - o *bit* mais significativo de v é representado por v_{n-1},
      - v é representado *bit* a *bit* por $(v_{n-1}\ v_{n-2}\ \ldots\ v_2\ v_1\ v_0)$.
    - uma palavra formada pelos *bits* de índices i até j é representada por $v_{j..i}$.
    - para facilitar a representação de palavras de 4 *bits* são usadas as letras maiúsculas. As palavras de 4 *bits* também são numeradas da palavra menos significativa para a mais significativa. Logo, V_i representa a palavra $v_{4i+3..4i}$.
    - para representar um *bit* de uma palavra de 4 *bits* é utilizado um sub-índice. Por exemplo, $(V_j)_i$ representa o *bit* v_{4j+i}.

- Concatenação de *bits*:
  - Sejam dois números x e y, onde x é representado por n *bits* e y é representado por m *bits*. Seja z a concatenação de x e y. A concatenação z é representada por $n+m$ *bits* e a fórmula para o seu cálculo é $z = 2^m x + y$. A notação para a concatenação z é (x, y).

- Representação de uma soma:
  - Seja a soma $z = x + y$, onde x e y são representados por n *bits*.
    - o resultado z tem que ser representado com $n+1$ *bits* para que não haja risco de *overflow*. Sendo assim, a soma é representada por $z_{n..0} = x_{n-1..0} + y_{n-1..0}$.

- o resultado z também pode ser representado com n *bits* mais um *bit* de *carry*. Desta forma, a soma é representada por $(c, z_{n-1\cdots 0}) = x_{n-1\cdots 0} + y_{n-1\cdots 0}$.
- analisando a soma *bit* a *bit*, é convencionado que c_i representa o sinal de *carry* resultante da soma $x_{i-1} + y_{i-1} + c_{i-1}$. Esta soma de um *bit* é representada por $(c_i, z_{i-1}) = x_{i-1} + y_{i-1} + c_{i-1}$.

- Atribuições e comparações - para distinguir uma operação de atribuição de uma comparação é utilizada a mesma notação da linguagem C, ou seja, o símbolo '=' representa uma atribuição e o símbolo '= =' representa uma comparação. Por exemplo, $a = b$ significa que o valor da variável *b* está sendo atribuído à variável *a*, enquanto que $a == b$ significa que os conteúdos das variáveis *a* e *b* estão sendo comparados.

III.2 - Representação em Complemento a Dois

Para os números inteiros do PRIMATA, foi escolhida a representação em complemento a dois. Antes de começar a analisar os problemas que envolvem a implementação das quatro operações básicas, cabe fazer uma pequena revisão a respeito das várias formas de representação, a fim de melhor justificar a escolha feita.

São três os sistemas de numeração posicional mais conhecidos: sinal e magnitude, complemento a um e complemento a dois. Um número inteiro x é representado com n *bits* em cada um destes sistemas binários da seguinte forma:

a) Sinal e magnitude:
- Os $n-1$ *bits* menos significativos contém a magnitude, ou seja $|x|$.
- O *bit* mais significativo contém o sinal (0 se $x \geq 0$ ou 1 se $x \leq 0$).
- Intervalo de representação: $[-(2^{n-1}-1), 2^{n-1}-1]$.

b) Complemento a um:
- Se $x \geq 0$, então x é representado por $|x|$,
- Se $x \leq 0$, então x é representado por $(2^n - 1) - |x| = (11..111) - |x| = \sum_{i=0}^{n-1} (1 - |x|_i) 2^i$, o que equivale a complementar $|x|$, *bit* a *bit*. Ou seja, x é representado por $\overline{|x|}$.
- Intervalo de representação: $[-(2^{n-1}-1), 2^{n-1}-1]$.

c) Complemento a dois:
- Se $x \geq 0$, então x é representado por $|x|$
- Senão $(x < 0)$, então x é representado por $2^n - |x|$. Do ponto de vista prático: $2^n - |x| = (2^n - 1) - |x| + 1 = \overline{|x|} + 1$. Logo, a representação de x pode ser encontrada complementando $|x|$, *bit* a *bit* e somando um ao resultado da complementação.
- Intervalo de representação: $[-2^{n-1}, 2^{n-1}-1]$.

Para todas os sistemas apresentados, reconhecer se um número é positivo ou negativo é uma tarefa muito simples, bastando para isso testar o seu *bit* mais significativo.

A "troca de sinal" na representação em sinal e magnitude é a mais simples, bastando para isso complementar o *bit* mais significativo do número. No segundo sistema, todos os *bits* são complementados. A representação em complemento a dois, além de complementar todos os *bits*, exige ainda que seja somado 1 (um) ao *bit* menos significativo.

Por outro lado, a representação em complemento a dois tem a vantagem de possuir uma única representação para o número 0 (zero), enquanto que os outros sistemas possuem duas representações diferentes para este número.

Mas a principal razão pela qual a maioria dos computadores utiliza a representação em complemento a dois é a facilidade que este sistema oferece na execução das operações de soma e subtração, em relação aos outros sistemas. Esta vantagem é ampliada ainda mais se forem consideradas as operações de multiplicação e divisão, já que as suas execuções são compostas por séries de somas e subtrações. Abaixo é apresentada uma breve análise da soma $s = x + y$ para cada um dos sistemas. Esta análise pode ser facilmente estendida a operação de subtração.

a) Sinal e magnitude:
- Inicialmente os sinais dos operandos devem ser comparados.
- Se os operandos tem o mesmo sinal, então
  - o sinal de *s* é o mesmo dos operandos (*x* e *y*)
  - a magnitude de *s* é dada pela soma das magnitudes de *x* e *y*.
  - se a soma das magnitudes exceder $2^{n-1} - 1$ (sinal de *carry* $c_{n-1} = 1$), então *s* não pode ser representado por *n bits* (*overflow*).
- senão (os operandos têm sinais opostos), então
  - compara as magnitudes dos operandos
  - o sinal de *s* é o mesmo do operando de maior magnitude
  - a magnitude de *s* é dada pela subtração do operando de maior magnitude pelo de menor magnitude.

b) Complemento a um:
- Inicialmente soma-se os *n bits* dos dois operandos e coloca-se o resultado em *s*.
- A partir daí a análise pode ser dividida em três casos:

b.1) os dois operandos são positivos (*bits* mais significativos iguais a 0):
- Então $s = |x| + |y|$ e o sinal de *carry* c_n é igual a 0
- Se o *bit* mais significativo de *s* é igual a 1 (um), então *s* não pode ser representado por *n bits* (*overflow*).

b.2) os operandos têm sinais opostos. Considerando-se, sem perda de generalidade, que x é negativo ($x_{n-1}=1$) e y é positivo ($y_{n-1}=0$), tem-se então: $s=2^n-1-|x|+|y|=2^n-1-(|x|-|y|)$. A partir daí surgem dois subcasos:

b.2.1) $|x|\geq|y|$.

- Isto significa que $(|x|-|y|)\geq 0$, logo o sinal de *carry* c_n é igual a 0 (zero)
- O resultado correto, $s=2^n-1-(|x|-|y|)$, já está armazenado em *s*.

b.2.2) $|x|<|y|$.

- Isto significa que $(|x|-|y|)<0$, logo o sinal de *carry* c_n é igual a 1 (um)
- O resultado correto é $s=|y|-|x|$. Como, numa representação de *n bits*, o termo 2^n não é aproveitado, para obter o resultado correto basta somar 1 (um) a *s*.

b.3) os dois operandos são negativos:

- Então $s=2^n-1-|x|+2^n-1-|y|=2^{n+1}-2-(|x|+|y|)$
- Isto significa que o sinal de *carry* c_n é igual a 1 (um)
- O resultado correto é $s=2^n-1-(|x|+|y|)$. Como, numa representação de *n bits*, os termos 2^{n+1} e 2^n são equivalentes (ambos não são aproveitados), para obter o resultado correto basta somar 1 (um) a *s*.
- Se o *bit* mais significativo de *s* é igual a 0 (zero), então *s* não pode ser representado por *n bits* (*overflow*).

- Em resumo, para todos os casos, para corrigir o resultado, basta somar *carry* c_n ao *bit* menos significativo de *s*.

c) Complemento a dois:

- Basta somar os *n bits* dos dois operandos e colocar o resultado em *s*. A prova disto é dividida em três casos:

c.1) os dois operandos são positivos ou nulos:

- Então $s=|x|+|y|$.
- Se o *bit* mais significativo de *s* é igual a 1 (um), então *s* não pode ser representado por *n bits* (*overflow*).

c.2) um operando é positivo ou nulo e o outro é negativo. Considerando-se, sem perda de generalidade, que *x* é negativo e *y* é positivo ou nulo, tem-se então dois subcasos:

c.2.1) $|x|\geq|y|$.

- $s=2^n-|x|+|y|=2^n-(|x|-|y|)\equiv -|x|+|y|$. que é o resultado correto.

c.2.2) $|x|<|y|$.

- $s=2^n-|x|+|y|=2^n+|y|-|x|$. Como, numa representação de *n bits*, o termo 2^n não é aproveitado, então $s=|y|-|x|$, que é o resultado correto.

c.3) os dois operandos são negativos:

- Então $s = 2^n - |x| + 2^n - |y| = 2^{n+1} - (|x| + |y|)$
- O resultado correto é $s = 2^n - (|x| + |y|)$. Como, numa representação de *n bits*, os termos 2^{n+1} e 2^n são equivalentes (ambos não são aproveitados), então *s* contém o resultado correto.
- Se o *bit* mais significativo de *s* é igual a 0 (zero), então *s* não pode ser representado por *n bits* (*overflow*).

Em resumo, o pior sistema para a execução de somas é o de representação em sinal e magnitude, seguido do sistema de representação em complemento a um. O melhor sistema é o de representação em complemento a dois. O primeiro sistema é o mais lento, já que, além da soma (ou subtração) de magnitudes e da comparação de sinais, ele pode necessitar de uma comparação de magnitudes. A execução de tarefas diferentes de acordo com os operandos torna a implementação deste sistema mais complexa para uma arquitetura SIMD. Para este tipo de arquitetura, todas as operações se tornam obrigatórias, aumentando o tempo de execução da soma.

O segundo sistema apresenta como desvantagem a possível necessidade de uma segunda soma, dependendo do resultado da primeira (se $c_n = 1$). Como se trata de uma arquitetura SIMD, esta segunda soma sempre será executada, aumentando o tempo de execução da soma.

Já no sistema de representação em complemento a dois, a soma é a mais simples possível, como se tratasse de uma soma entre dois números inteiros sem sinal. A única diferença encontra-se no teste de *overflow*, que tem que levar em consideração os sinais dos operandos.

Sendo assim, da mesma forma que a grande maioria dos computadores atuais, o PRIMATA tem seus números inteiros representados em complemento a dois.

Nas próximas seções é apresentado um estudo das alternativas para a implementação das quatro operações aritméticas básicas: soma, subtração, multiplicação e divisão. A partir deste estudo, é deduzida uma proposta de arquitetura para o EP.

III.3 - Soma de Inteiros

Considere a soma $z = x + y$, onde *x*, *y* e *z* são números inteiros representados em complemento a dois com *n bits*. Como foi apresentado na seção anterior, uma soma simples dos *n bits* dos dois operandos resolve corretamente a operação. Resta analisar qual é a arquitetura mais adequada para realizar esta operação.

Para realizar uma soma, a arquitetura básica de um EP 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 estão armazenados os operandos e o resultado. A Unidade de Controle (UC) tem que conter: três bases de endereço (duas para os operandos e uma para o resultado), um contador (para indexar as bases de endereço) e um comparador (para controlar o *loop*).

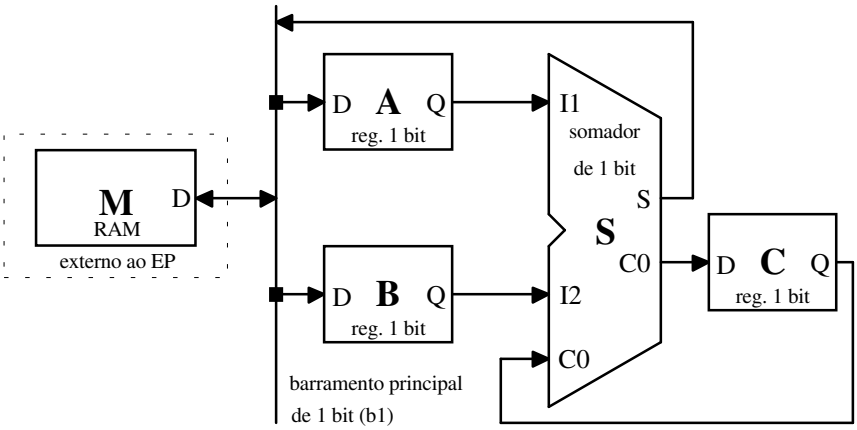

Figura III.1 - EP básico de 1 *bit* para soma

Considere, então, um EP básico de 1 *bit* composto por: três registradores (A, B e C), um somador (S) e associado a uma memória (M), como mostra a Figura III.1. Considere também que a UC possui um contador (I) e que as suas bases de endereço contêm o endereço do *bit* menos significativo de cada um dos operandos e do resultado na 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 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 de 4 *bits*, como mostra a Figura III.2. Para a arquitetura apresentada, 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 a soma: $3n/4 + 1$.

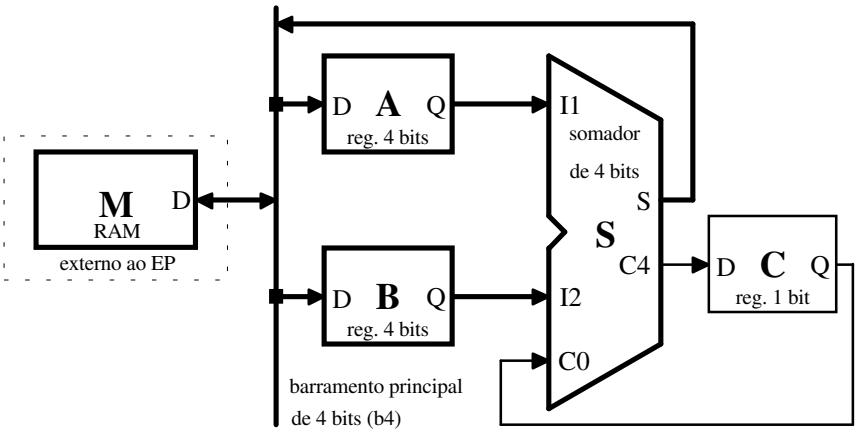

Figura III.2 - EP básico de 4 *bits* para soma

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

A otimização dos acessos aos dados pode ser feita através da implementação de uma memória interna ao EP, do tipo *double port* ou *triple port*. Esta solução aumenta o custo do processador. Por outro lado, o uso de uma memória *double port*, torna desnecessária a utilização de dois registradores para o armazenamento dos operandos. Pode-se então construir um EP idêntico ao primeiro, porém com uma memória *double port* e apenas dois registradores de 1 *bit* (A e C), como mostra a Figura III.3. Para esta arquitetura, o algoritmo que resolve a soma é:

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

Número de ciclos necessários para a soma: $2n + 1$

Já o uso de uma memória *triple port* torna desnecessária a utilização de qualquer registrador para o armazenamento de operandos. Para um EP idêntico ao primeiro, porém com uma memória *triple port* e com apenas um registrador (C), como mostra a Figura III.4, o algoritmo que resolve a soma é:

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

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

As duas soluções apresentadas acima diminuem cerca de duas (*double port*) a três vezes (*triple port*) o tempo de execução da soma, mas a um custo bem superior ao da primeira solução (EP de 4 *bits*). Para se ter uma idéia, a implementação [Beal91] de uma memória *triple port* de 256 *bits* (que não é um valor muito elevado) gasta cerca de 3100 transistores, enquanto que toda uma Unidade Lógica e Aritmética de 4 *bits* gasta cerca de 500 transistores. 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.

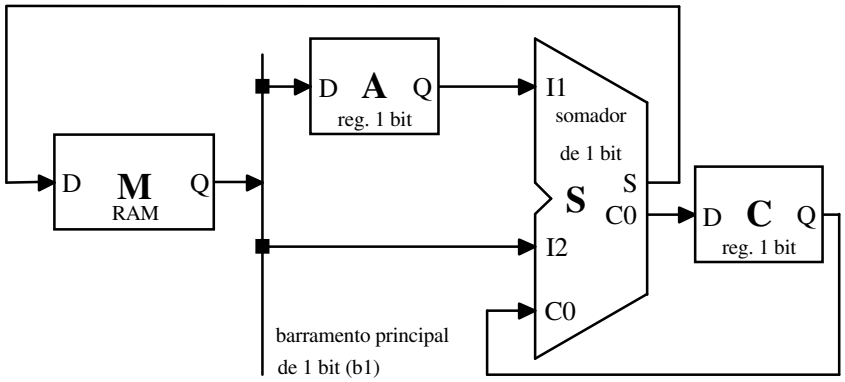

Figura III.3 - EP de 1 *bit* com memória *double port* para soma

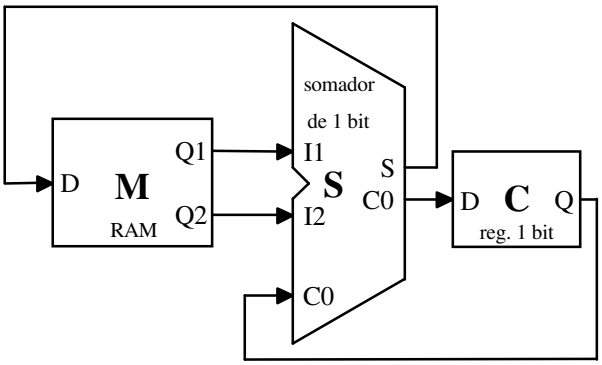

Figura III.4 - EP de 1 *bit* com memória *triple port* para soma

III.4 - Subtração de Inteiros

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, como foi deduzido na seção III.2, $-y = \overline{y} + 1$. 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.

Através de uma pequena alteração na arquitetura apresentada para a soma, é possível executar essas duas tarefas (soma e complementação), simultaneamente.

Considere, então, o EP básico de 1 *bit* apresentado na seção anterior, acrescido de uma unidade lógica (L), capaz de complementar a sua entrada, como mostra a Figura III.5. Para esta arquitetura, o algoritmo que resolve a subtração 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 a subtração: $3n + 1$.

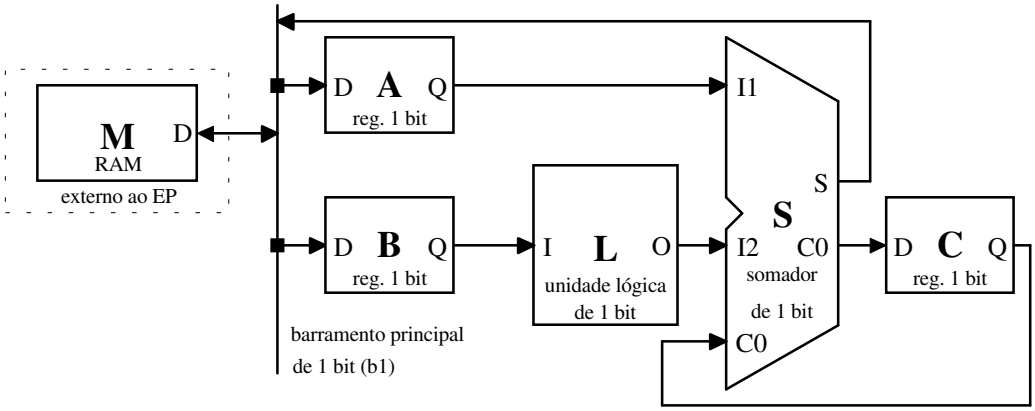

Figura III.5 - EP básico de 1 *bit* para subtração

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.

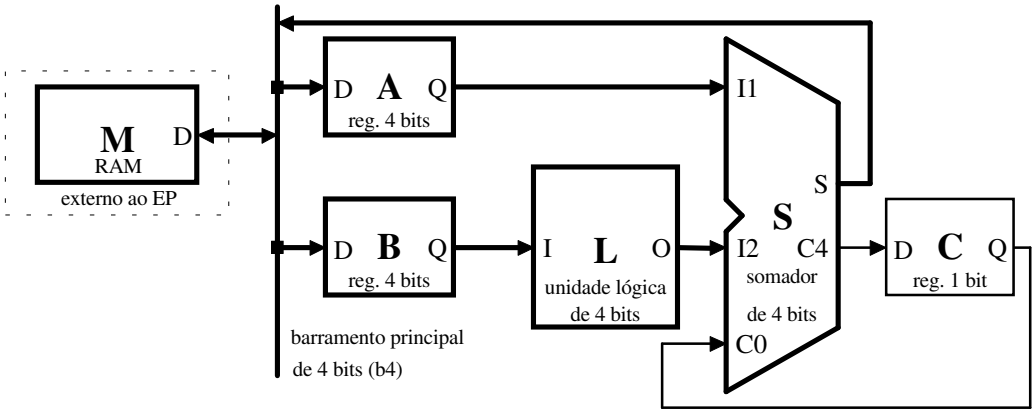

Figura III.6 - EP básico de 4 *bits* para subtração

Sendo assim, pode-se construir uma arquitetura idêntica a primeira, porém utilizando um EP com registradores, unidade lógica, somador e memória com largura de 4 *bits* (Figura III.6). Para esta arquitetura, 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 a subtração: $3n/4 + 1$.

Por esta solução, a área do processador é praticamente quadruplicada. 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.

III.5 - Multiplicação de Inteiros

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

III.5.1 - Multiplicação de Dois Inteiros Positivos

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 $2n$ *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 $2n$ *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 $2n$ *bits* e n somas de n *bits* com armazenamento do *carry*.

III.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.

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 por 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:

$$p = (2^{2n} - |x|) \times (2^n - |y|) - 2^n(2^{2n} - |x|) =$$
$$p = 2^{3n} - 2^{2n}|y| - 2^n|x| + |x \times y| - 2^{3n} + 2^n|x| =$$
$$p = |x \times y| - 2^{2n}|y| = |x \times y| \equiv x \times y$$

- subtração dos dois termos em excesso, levando a:

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

Tabela III.1 - Correções para a multiplicação de inteiros com sinal

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

A Tabela III.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 de sinal de *x* de *n* para 2*n bits*. Analisando a Tabela III.1 é possível deduzir dois métodos gerais para a multiplicação de dois números inteiros representados em complemento a dois.

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 2*n 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. Da forma como ele está apresentado, são necessárias: uma atribuição de 2*n bits*, *n* somas de 2*n bits* cada uma e uma subtração de 2*n bits*.

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. Ambas têm o mesmo fator condicionante, ou seja, $y_{n-1} = 1$. Neste caso, serão executadas as seguintes operações:

- $p = p + 2^{n-1} x'$
- $p = p - 2^n x'$

Juntando as duas operações tem-se:

$$p = p + 2^{n-1} x' - 2^n x' = p + 2^{n-1}(x' - 2x') = p - 2^{n-1} x'$$

Por último, as somas e a subtração não precisam ser de 2*n 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*. Intuitivamente é fácil compreender esta simplificação. O número a ser somado a p é sempre x', apenas deslocado de alguns *bits*, o que implica que o número somado tem sempre um mesmo número máximo de algarismos significativos, que é $n-1$. Os demais *bits* representam apenas o sinal e o deslocamento. A cada soma, como x' é deslocado mais uma posição à esquerda, p passa a ter mais um *bit* significativo e necessita de mais um *bit* para representar o seu sinal.

Para provar esta simplificação será utilizado o método da indução. Primeiramente, deve-se considerar que x' tem no máximo $n-1$ *bits* significativos, já que x é representado por *n bits*. Os demais *bits* são todos iguais e representam o sinal de x'. Sendo assim, pode-se dizer que: $x'_{2n-1} = x'_{2n-2} = \cdots = x'_{n-1} = s$, onde s é o sinal de x'.

No início do algoritmo, p é igualado a zero.

Para $i = 0$, tem-se que:

- se $y_0 == 0$, então $p = p + 0 = 0$
- se $y_0 == 1$, então $p = p + x' = x'$

Logo, após o primeiro passo, p tem no máximo o mesmo número de *bits* significativos que x', ou seja, $n-1$. Os demais *bits* são iguais e representam o sinal.

Supondo agora que ao final da iteração de ordem k ($i = k$), p tenha no máximo $k+n$ *bits* significativos, sendo que os demais *bits* são iguais e representam o sinal, pode-se dizer que: $p_{2n-1} = p_{2n-2} = \cdots = p_{k+n} = t$, onde t é o sinal de p após a iteração de ordem k.

Para $i = k+1$, tem-se que:

- se $y_{k+1} == 0$, então $p = p + 0 = p$
- se $y_{k+1} == 1$, então $p = p + 2^{k+1} x'$

No primeiro caso, o valor de p não é alterado, logo ele continuará a possuir no máximo $n+k$ *bits* significativos.

No segundo caso, a execução da soma é dada por:

$c_0 = 0$

para j variando de 0 até k

$(c_{j+1}, p_j) = p_j + 0 + c_j$

para j variando de $k+1$ até $2n-1$

$(c_{j+1}, p_j) = p_j + x'_{j-k-1} + c_j$

Simplificando:

$c_{k+1} = 0$

para j variando de $k+1$ até $k+n-1$

$(c_{j+1}, p_j) = p_j + x'_{j-k-1} + c_j$

para j variando de $k+n$ até $2n-1$

$(c_{j+1}, p_j) = t + s + c_j$

O primeiro laço equivale a uma soma de $n-1$ *bits*. Já do segundo laço pode-se tirar que:

$c_{k+n+1} = s.t + s.c_{k+n} + t.c_{k+n}$

$c_{k+n+2} = s.t + s.c_{k+n+1} + t.c_{k+n+1} = s.t + (s+t).(s.t + s.c_{k+n} + t.c_{k+n}) =$

$c_{k+n+2} = s.t + s.t + s.c_{k+n} + s.t.c_{k+n} + s.t + s.t.c_{k+n} + t.c_{k+n} = s.t + s.c_{k+n} + t.c_{k+n} =$

$c_{k+n+2} = c_{k+n+1}$

$c_{k+n+3} = s.t + s.c_{k+n+2} + t.c_{k+n+2} = s.t + s.c_{k+n+1} + t.c_{k+n+1} =$

$c_{k+n+3} = c_{k+n+2} = c_{k+n+1}$

...

Logo, $c_{2n-1} = c_{2n-2} = \cdots = c_{k+n+2} = c_{k+n+1}$

Além disso:

$$p_{k+n} = s \oplus t \oplus c_{k+n}$$
$$p_{k+n+1} = s \oplus t \oplus c_{k+n+1}$$
$$p_{k+n+2} = s \oplus t \oplus c_{k+n+2} = s \oplus t \oplus c_{k+n+1} = p_{k+n+1}$$
$$p_{k+n+3} = s \oplus t \oplus c_{k+n+3} = s \oplus t \oplus c_{k+n+1} = p_{k+n+1}$$
...

Logo, $p_{2n-1} = p_{2n-2} = \cdots = p_{k+n+2} = p_{k+n+1} = s \oplus t \oplus c_{k+n+1}$

Sendo assim, após a iteração de ordem $k+1$ ($i = k+1$), p terá, no máximo, $k+n+1$ *bits* significativos. Os demais *bits* são iguais e representam o sinal de p.

Em resumo, foi provado que, para $i = 0$:

- é suficiente uma soma de n *bits*.
- ao final da iteração, o número de *bits* significativos de p é menor ou igual a $i + n = n$.

Supondo que ao final da iteração de ordem k ($i = k$), o número de *bits* significativos de p é menor ou igual a $i + n = k + n$, então, para $i = k + 1$:

- é suficiente uma soma de n *bits* acrescida do cálculo do novo sinal de p.
- ao final da iteração, o número de *bits* significativos de p é menor ou igual a $i + n = k + n + 1$.

Fica então provado por indução que não são necessárias somas de $2n$ *bits*, mas apenas somas de n *bits* acrescidas do cálculo do novo sinal de p.

Após as modificações propostas, o algoritmo de multiplicação fica:

se $y_0 == 1$, então
    $p_{n-1..0} = x'_{n-1..0}$
senão
    $p_{n-1..0} = 0$
para i variando de 1 até $n-2$
    $p_{n+i} = p_{n+i-1}$
    se $y_i == 1$, então
        $p_{n+i-1..i} = p_{n+i-1..i} + x'_{n-1..0}$
$p_{2n-1} = p_{2n-2}$
se $y_{n-1} == 1$, então
    $p_{2n-1-1..n-1} = p_{2n-1-1..n-1} - x'_{n-1..0}$

Para executar este algoritmo são necessárias: n comparações de 1 *bit*, uma atribuição de $n+1$ *bits*, $n-1$ 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 em *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 pode ser bastante melhorado. Da forma como ele está apresentado, são necessárias: $n+2$ comparações de 1 *bit*, uma atribuição de 2*n bits*, *n* somas de *n bits* com armazenamento do *carry* e duas subtrações de *n bits* sendo que uma com armazenamento do *carry*.

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 do termo em excesso em *x* podem ser condensadas em uma única operação. Ambas têm o mesmo fator condicionante, ou seja, $y_{n-1} = 1$. Neste caso, serão executadas as seguintes operações:

- $p = p + 2^{n-1} x$
- $p = p - 2^n x$

Juntando as duas operações tem-se:

$$p = p + 2^{n-1} x - 2^n x = p + 2^{n-1}(x - 2x) = p - 2^{n-1} x$$

Com as modificações propostas acima, o algoritmo de multiplicação fica:

se $y_0 == 1$
    $p_{n-1 \cdot\cdot 0} = x_{n-1 \cdot\cdot 0}$
senão
    $p_{n-1 \cdot\cdot 0} = 0$
$p_n = 0$
para *i* variando de 1 até $n-2$
    se $y_i == 1$
        $p_{n+i \cdot\cdot i} = p_{n+i-1 \cdot\cdot i} + x_{n-1 \cdot\cdot 0}$
se $y_{n-1} == 1$
    $p_{2n-1 \cdot\cdot n-1} = p_{2n-2 \cdot\cdot n-1} - x_{n-1 \cdot\cdot 0}$
se $x_{n-1} == 1$
    $p_{2n-1 \cdot\cdot n} = p_{2n-1 \cdot\cdot n} - y_{n-1 \cdot\cdot 0}$

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, como será visto adiante.

Considere um processador que trabalha com palavras de 1 *bit*. O algoritmo apresentado para o primeiro método executa $n^2 + 3n - 1$ operações de 1 *bit* contra $n^2 + 3n + 1$ operações de 1 *bit* do algoritmo apresentado para o segundo método. Esta análise inicial induz a uma pequena vantagem do primeiro método. Esta vantagem se amplia se for analisada uma implementação típica de um processador de 1 *bit*, em que existe um registrador para operações condicionais *K*, um registrador para o *carry C* e registradores para o armazenamento de operandos *A* e *B*. Para um processador com estas características, o algoritmo de multiplicação para o 1º método pode ser ligeiramente modificado, resultando em:

$K = y_0$
para i variando de 0 até $n-1$
    $B = x_i$
    $p_i = B \; and \; K$
$p_n = B$ and K
para j variando de 1 até $n-2$
    $K = y_j, C = 0$
    para i variando de 0 até $n-1$
        $A = p_{i+j}$
        $B = x_i$
        $(C, p_{i+j}) = A + B \; and \; K + C$
    $p_{j+n} = A + B \; and \; K + C$
$C = K = y_{n-1}$
para i variando de 0 até $n-1$
    $A = p_{i+n-1}$
    $B = x_i$
    $(C, p_{i+n-1}) = A + \overline{B} \; and \; K + C$
$p_{2n-1} = A + \overline{B} \; and \; K + C$

Este novo algoritmo elimina as $n-1$ atribuições de 1 *bit*, presentes no algoritmo inicial. Desta forma, são necessárias apenas $n^2 + 2n$ operações de 1 *bit* para a execução da multiplicação, o que torna o 1º método mais eficiente do que o 2º método para este tipo de processadores.

Para analisar os processadores de 4 *bits*, será considerado que o tamanho dos operandos da multiplicação é um múltiplo de 4. Na prática, esta restrição é uma realidade na maior parte dos casos.

Considere agora 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 4), explorando melhor e portanto se adaptando melhor aos processadores de 4 *bits*.

III.5.3 - EP Básico de 1 *Bit* para a Multiplicação

Para implementar a multiplicação pode-se adaptar a máquina apresentada para a solução da subtração (Figura III.5), acrescentando um registrador para controle condicional local (K), como mostra a Figura III.7. O desempenho do algoritmo desenvolvido (Apêndice A) a partir do 1º método ainda não é o ideal. Ele processa a multiplicação em $3n^2+n$ ciclos. Este tempo é facilmente explicado. São executadas n operações (uma atribuição, $n-2$ somas e uma subtração) envolvendo $n+1$ *bits*. A atribuição demanda $2n+1$ ciclos para sua execução. Cada soma (ou subtração) necessita de $3n+1$ ciclos para a sua execução. 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. Sendo assim, para as $n-1$ operações de soma e subtração são necessários $3n^2-2n-1$ ciclos. Os demais ciclos são gastos na leitura do multiplicador y (n ciclos).

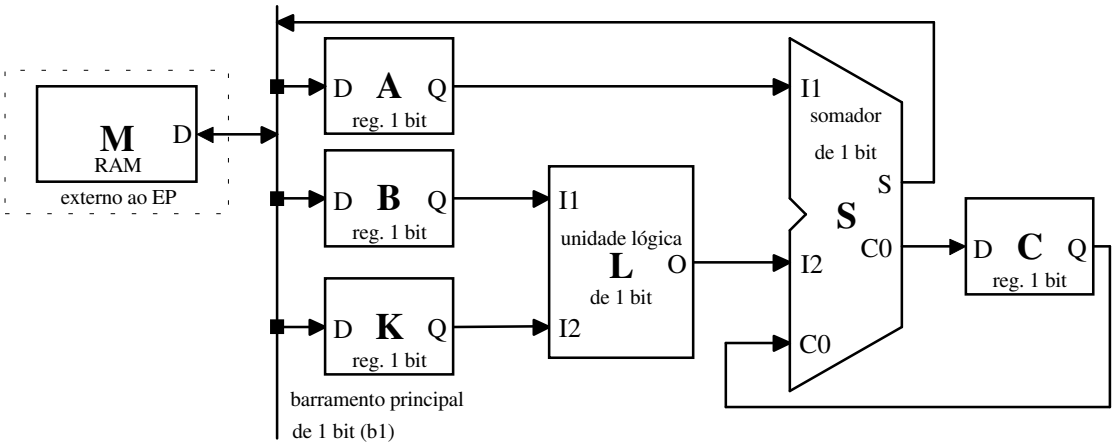

Figura III.7 - EP básico de 1 *bit* para a multiplicação

III.5.4 - EPs de 1 *Bit* com Memória Interna para a Multiplicação

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*. São apresentadas duas implementações para esta memória interna: a primeira é

um registrador de deslocamento de tamanho programável (solução adotada pelo MPP [Batc80]) e a segunda é uma fila *FIFO* (*First In First Out*).

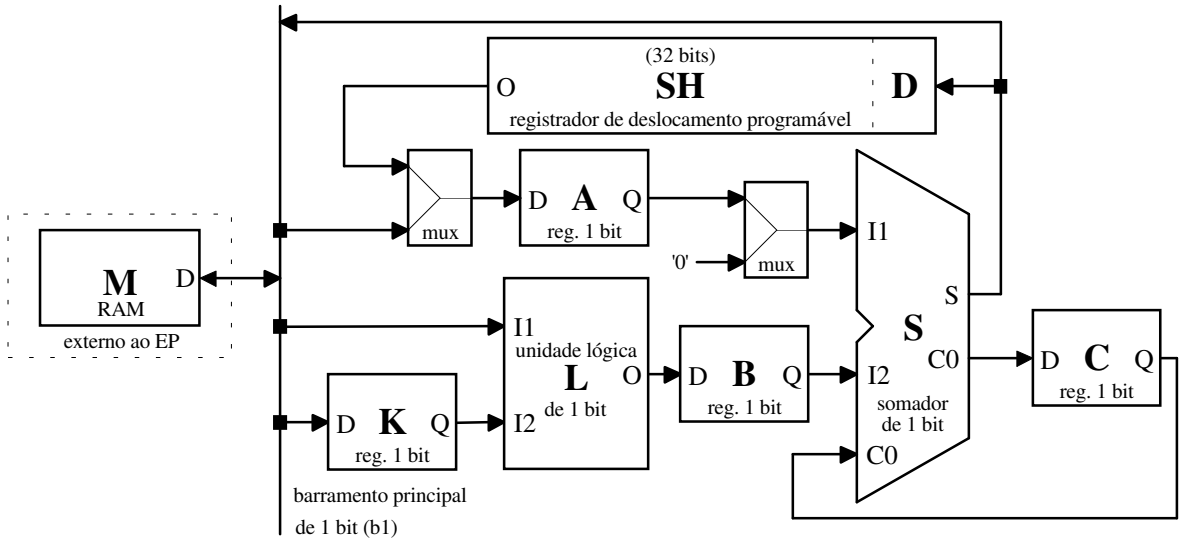

Figura III.8 - EP de 1 *bit* com registrador de deslocamento programável para a multiplicação

A Figura III.8 apresenta o esquema de um processador que utiliza um registrador de deslocamento de tamanho programável (SH) para o armazenamento de resultados parciais. Este registrador tem o *bit* mais significativo denominado D e o menos significativo ligado a entrada de A . Para que este processador execute a multiplicação deve-se configurar o registrador de deslocamento SH com tamanho $n-1$. O algoritmo desenvolvido é apresentado no apêndice A.

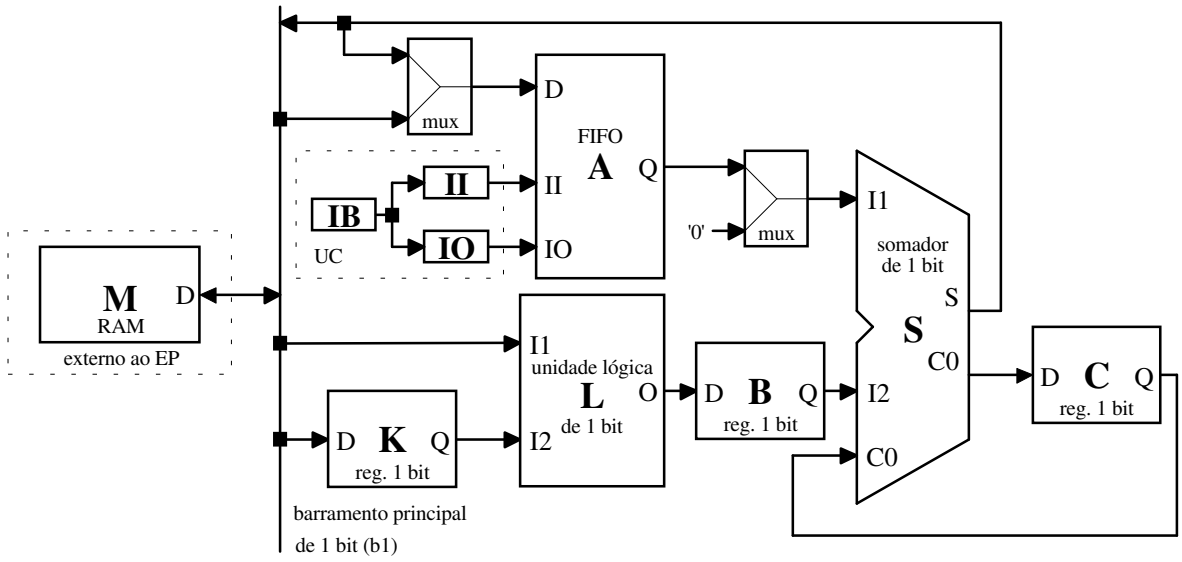

Figura III.9 - EP de 1 *bit* com fila FIFO para a multiplicação

A Figura III.9 apresenta o esquema de um processador que utiliza como memória interna um dispositivo de armazenamento similar a uma fila do tipo FIFO (*First In First Out*) de 32

bits denominada A. Esta fila tem por finalidade armazenar resultados parciais. Três índices de acesso à fila A, todos de 4 *bits*, são gerados pela Unidade de Controle (UC): II (índice de escrita), IO (índice de leitura) e IB (índice base). A fila A é um pouco mais flexível do que uma FIFO. Os valores iniciais de II e IO são independentes e carregados a partir de IB. Além disso, o pós-incremento de II nas operações de escrita e de IO nas operações de leitura não é automático, o que permite ler ou escrever em uma posição da fila mais de uma vez seguida. Os índices II e IO podem ser incrementados ao final de cada instrução. No caso de "estouro" de um dos índices, o mesmo retorna ao valor 0 (zero), o que torna a fila circular. O algoritmo de multiplicação desenvolvido desenvolvido para este processador é apresentado no apêndice A.

O desempenho das duas arquiteturas apresentadas acima é idêntico e bem melhor do que o obtido pelo EP básico de 1 *bit*. Ambas processam a multiplicação em $n^2 + 3n$ ciclos. São executadas n operações (uma atribuição, $n-2$ somas e uma subtração) envolvendo $n+1$ *bits*, o que demanda um total de $n^2 + n$ ciclos. Os demais ciclos são gastos na escrita do resultado (p) na memória ($2n$ ciclos).

III.5.5 - EP Básico de 4 *Bits* para a Multiplicação

Uma outra forma de melhorar o desempenho do EP na multiplicação é adotar a mesma solução encontrada para a soma e a subtração, ou seja, a utilização de um EP que trabalhe com dados de 4 *bits*.

Traduzindo o algoritmo o 2º método para um EP que processa dados de 4 *bits*, contendo dois registradores de 4 *bits* (A e B) para o armazenamento dos operandos, um registrador para o *carry* (C), e um registrador para as operações condicionais (K), chega-se a:

$K = y_0$
para i variando de 0 até $n/4 - 1$
    $B = x_{4i+3..4i}$
    $p_{4i+3..4i} = B \; and \; K$
$p_n = 0, j = 0, h = 1$
enquanto $4j + h < n - 1$
    $K = y_{4j+h}, C = 0$
    para i variando de 0 até $n/4 - 1$
        $A = p_{4(i+j)+h+3..4(i+j)+h}$
        $B = x_{4i+3..4i}$
        $(C, p_{4(i+j)+h+3..4(i+j)+h}) = A + B \; and \; K + C$
    $p_{4j+h+n} = A + B \; and \; K + C, h = h + 1$
    se $h < 3,$ então $h = 0, j = j + 1$
$C = K = y_{n-1}$