UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MILENE HÄNDEL

Circuitos Aritméticos e Representação Numérica por Resíduos

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

Prof. Dr. Renato Perez Ribas
Orientador

Prof. Dr. André Inácio Reis
Co-orientador

Porto Alegre, junho de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Händel, Milene

Circuitos Aritméticos e Representação Numérica por Resíduos / Milene Händel – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

49 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientador: Renato Perez Ribas; Co-orientador: André Inácio Reis.

1.RNS 2.Circuitos Aritméticos 3.Sistemas de Numeração. I. Ribas, Renato Perez. II. Reis, André Inácio. III. Título.

AGRADECIMENTOS

Agradeço ao meu orientador, Renato Perez Ribas, meu Co-orientador, André Inácio Reis, meus colegas do Nangate Research Lab, principalmente a Paula Karina Perez Vieira, o Felipe Marranghello e o Giovani Heriberto Sartori pela ajuda na síntese no Synopsys e Alessandro Goulart pelos diagramas, e todos que me apoiaram neste trabalho. Faço um agradecimento especial ao Eduardo, que tem compartilhado comigo todas as dificuldades e alegrias destes últimos anos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS.....6

LISTA DE FIGURAS.....7

LISTA DE TABELAS.....9

RESUMO.....11

ABSTRACT.....12

1 INTRODUÇÃO.....13

2 SISTEMAS DE REPRESENTAÇÃO.....15

2.1 Sistema Convencional.....15
2.2 Sistema Signed-Digit.....16
2.2.1 Conversão de notação convencional para signed-digit.....16
2.2.2 Conversão de notação signed-digit para convencional.....17
2.3 Sistema Logarítmico.....17
2.3.1 Logaritmos.....17
2.4 Sistema RNS.....19
2.4.1 Escolha da Base.....20

3 OPERADORES ARITMÉTICOS.....22

3.1 Somadores.....22
3.1.1 Somador Ripple Carry.....22
3.1.2 Somador Carry Select.....23
3.1.3 Somador Carry Skip.....25
3.1.4 Somador Carry Lookahead.....25
3.1.5 Somadores de Prefixo Paralelos.....25
3.2 Multiplicadores.....27
3.2.1 Multiplicação Seqüencial.....28
3.2.2 Multiplicação Paralela.....29
3.2.3 Multiplicação Seqüencial X Paralela.....33

4 OPERADORES RNS.....34

4.1 Codificação de notação convencional para RNS.....35
4.2 Decodificação de RNS para notação convencional.....36

5 IMPLEMENTAÇÃO E RESULTADOS.....39

5.1 Operadores Aritméticos.....39
5.2 Circuitos RNS.....40
5.3 Ferramentas Utilizadas.....41
5.4 Resultados e Análise.....42
5.4.1 Somadores 16 bits.....42
5.4.2 Somadores de 32 bits.....43
5.4.3 Somadores de 64 bits.....44
5.4.4 Multiplicadores de 16 bits.....45
5.4.5 Multiplicadores de 32 bits.....46
5.4.6 Multiplicadores de 64 bits.....47
5.4.7 Análise Comparativa.....47
CONCLUSÃO.....49
REFERÊNCIAS.....50

LISTA DE ABREVIATURAS E SIGLAS

CLA	Carry Lookahead Adder
CRT	Chinese Reminder Theorem
CSA	Carry Select Adder
FIR	Finite Impulse Response
IP	Intellectual Property
LNS	Logarithmic Number System
MAC	Multiplica-Acumula
MDC	Máximo Divisor Comum
PPA	Parallel Prefix Adder
RCA	Ripple Carry Adder
RNS	Residue Number System
SD	Signed-Digit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

LISTA DE FIGURAS

FIGURA 2.1: CONVERSÃO DE UM NÚMERO PARA SIGNED-DIGIT 17

FIGURA 2.2 : GRÁFICO DE LOGARITMOS..... 18

FIGURA 3.3 : SOMADOR RIPPLE-CARRY.....23

FIGURA 3.4: SOMADOR CARRY SELECT.....24

FIGURA 3.5: SOMADOR CARRY SELECT DE 16 BITS.....24

FIGURA 3.6: SOMADOR PPA BRENT-KUNG.....26

FIGURA 3.7: SOMADOR PPA KOGGE-STONE.....27

FIGURA 3.8: SOMADOR PPA HÍBRIDO..... 27

FIGURA 3.9: EXEMPLO DE MULTIPLICAÇÃO DE NÚMEROS SEM SINAL NA BASE BINÁRIA..... 28

FIGURA 3.10: ÁRVORES DE WALLACE DE (A) 3 BITS E (B) 7 BITS..... 30

FIGURA 3.11: ARRAY DE MULTIPLICAÇÃO.....31

FIGURA 3.12: ALINHAMENTO DE QUATRO SUBPRODUTOS NAS FORMAS (A) DIRETA E (B) ORGANIZADAS NA ESTRUTURA MODULAR(KOREN, 2002)..... 32

FIGURA 3.13: ALINHAMENTO DE 16 PRODUTOS PARCIAIS (KOREN, 2002)..... 32

FIGURA 3.14: MULTIPLICADOR MODULAR ADITIVO DE 4 BITS.33

FIGURA 3.15: MULTIPLICADOR MODULAR ADITIVO DE 8 BITS.33

FIGURA 4.16: CODIFICADOR RNS (BI; JONES, 1988).....36

FIGURA 4.17: DECODIFICADOR RNS (BI; JONES, 1988).....38

FIGURA 5.18: ARQUITETURA DO SOMADOR CLA DE 64 BITS. .40

FIGURA 5.19: SIMULAÇÃO NO MODELSIM.....42

FIGURA 5.20: LEIAUTE DO SOMADOR CLA GERADO PELA FERRAMENTA DA SYNOPSYS.....42

LISTA DE TABELAS

TABELA 2.1: REPRESENTAÇÃO DE VALORES SEM SINAL EM DIFERENTES BASES.....16

TABELA 2.2: VALORES NO SISTEMA CONVENCIONAL NAS BASES BINÁRIA E DECIMAL E O EQUIVALENTE EM RNS NAS BASES (7|5|3) E (7|5|4).....19

TABELA 2.3:BASES RNS E SUAS ABRANGÊNCIAS (WANG ET AL., 2003).....21

TABELA 3.4: MULTIPLICAÇÃO NA BASE BINÁRIA.....28

TABELA 3.5: COMBINAÇÕES E ATRASOS DE ÁRVORES DE WALLACE.....30

TABELA 5.6: RESULTADOS SYNOPSYS PARA SÍNTESE DO SOMADOR DE 16 BITS.....43

TABELA 5.7: RESULTADOS XILINX DO SOMADOR DE 16 BITS. .43

TABELA 5.8: RESULTADOS SYNOPSYS PARA SÍNTESE DO SOMADOR DE 32 BITS.....44

TABELA 5.9: RESULTADOS XILINX DO SOMADOR DE 32 BITS. .44

TABELA 5.10: RESULTADOS SYNOPSYS PARA SÍNTESE DO SOMADOR DE 64 BITS.....44

TABELA 5.11: RESULTADOS XILINX DO SOMADOR DE 64 BITS 45

TABELA 5.12: RESULTADOS SYNOPSYS PARA SÍNTESE DO MULTIPLICADOR DE 16 BITS.....45

TABELA 5.13: RESULTADOS XILINX DO MULTIPLICADOR DE 16 BITS.....46

TABELA 5.14: RESULTADOS SYNOPSYS PARA SÍNTESE DO MULTIPLICADOR DE 32 BITS.....46

TABELA 5.15: RESULTADOS XILINX DO MULTIPLICADOR DE 32 BITS..... 46

TABELA 5.16: RESULTADOS SYNOPSYS PARA SÍNTESE DO MULTIPLICADOR DE 64 BITS..... 47

TABELA 5.17: RESULTADOS XILINX DO MULTIPLICADOR DE 64 BITS..... 47

RESUMO

Este trabalho mostra os diversos sistemas de representação numérica, incluindo o sistema numérico normalmente utilizado em circuitos e alguns sistemas alternativos. Uma maior ênfase é dada ao sistema numérico por resíduos. Este último apresenta características muito interessantes para o desenvolvimento de circuitos aritméticos nos dias atuais, como por exemplo, a alta paralelização.

São estudadas também as principais arquiteturas de somadores e multiplicadores. Várias descrições de circuitos aritméticos são feitas e sintetizadas. A arquitetura de circuitos aritméticos utilizando o sistema numérico por resíduos também é estudada e implementada. Os dados da síntese destes circuitos são comparados com os dados dos circuitos aritméticos tradicionais. Com isto, é possível avaliar as potenciais vantagens de se utilizar o sistema numérico por resíduos no desenvolvimento de circuitos aritméticos.

Palavras-Chave: RNS, sistema numérico por resíduos, sistemas numéricos, operadores aritméticos, somadores, multiplicadores.

Arithmetic Circuits and Residue Number System

ABSTRACT

This work shows various numerical representation systems, including the system normally used in current circuits and some alternative systems. A great emphasis is given to the residue number system. This last one presents very interesting characteristics for the development of arithmetic circuits nowadays, as for example, the high parallelization.

The main architectures of adders and multipliers are also studied. Some descriptions of arithmetic circuits are made and synthesized. The architecture of arithmetic circuits using the residue number system is also studied and implemented. The synthesis data of these circuits are compared with the traditional arithmetic circuits results. Then it is possible to evaluate the potential advantages of using the residue number system in arithmetic circuits development.

Keywords: RNS, residue number system, number systems, arithmetic operators, adders, multipliers.

1 INTRODUÇÃO

Os números têm um papel importante em sistemas computacionais. Números são a base e o objeto de operações de computação. A principal tarefa de computadores é realizar cálculos, trabalhando com números o tempo todo. Os humanos estão familiarizados com números há milhares de anos, entretanto a representação de números em computadores é um novo tópico. Um computador pode prover apenas dígitos finitos para a representação de números (tamanho fixo de palavra), enquanto um número real pode ser composto por infinitos dígitos. Devido às diferenças entre tamanho da palavra e tamanho do hardware, e entre atraso da propagação e precisão, vários tipos de representações numéricas foram propostas e adotadas.

Apesar da adição de números inteiros tipicamente ter o menor atraso de todas as funções aritméticas, ela têm o maior efeito no desempenho geral de um computador. Além da unidade inteira lógico-aritmética, todas as operações de ponto flutuante usam adição de inteiros em seus cálculos. Acessos à memória precisam de adições inteiras para a geração de seus endereços, e instruções de desvio usam adição para formar o endereço de instrução e para fazer comparações de valores. Todos esses diferentes usos sugerem que processadores modernos tipicamente contêm vários somadores inteiros, muitos dos quais podem aparecer em caminhos limitadores de freqüência.

Com o considerável progresso em tecnologia de circuitos VLSI (*Very Large Scale Integration*), muitos circuitos aritméticos impensáveis ontem se tornam componentes facilmente realizáveis hoje. Algoritmos que pareciam impossíveis de implementar agora têm possibilidades de implementação atrativas para o futuro. Isto significa que vale a pena investigar em novos projetos não somente os métodos de computação aritmética convencional, mas também os não convencionais.

As operações aritméticas mostram-se, portanto, como um ponto muito importante na implementação de circuitos, visto que seu custo e atraso são determinantes para o circuito como um todo. Por isso, se faz necessário um estudo aprofundado sobre a implementação dos operadores aritméticos, tendo em vista a simplificação dos circuitos e melhora de desempenho. Para isso, devem ser explorados seus algoritmos e representações numéricas (PARHAMI, 2000; LU, 2004).

Os sistemas numéricos alternativos vêm sendo bastante abordados em artigos recentes na área, como em (WEI, 2005), que mostra ser uma possível alternativa para os circuitos aritméticos desenvolvidos hoje em dia. O RNS se mostra uma opção interessante para aumentar o desempenho de operações, principalmente devido à redução das cadeias de *carry* nas operações de soma, subtração e multiplicação e à sua

grande possibilidade de paralelismo. Esta representação apresenta algumas limitações em operações como divisão, comparação de magnitude e detecção de *overflow*. Entretanto, em algumas operações de processamento digital de sinais, estas restrições podem gerar um impacto mínimo (MOHAN, 2002).

Neste trabalho, serão abordados, no capítulo 2, as principais características dos diferentes sistemas de representação numérica. O capítulo 3 mostra um estudo das principais arquiteturas de somadores e multiplicadores utilizadas atualmente no sistema binário. O capítulo 4 explora como são feitas as operações aritméticas no sistema numérico por resíduos, além de mostrar os princípios dos codificadores de binário para RNS e os decodificadores de RNS para binário. O capítulo 5 fala sobre as implementações feitas neste estudo e analisa os resultados obtidos. Para finalizar, são apresentadas as conclusões obtidas com a realização deste trabalho e as referências bibliográficas utilizadas.

2 SISTEMAS DE REPRESENTAÇÃO

Um computador pode fornecer apenas um número finito de dígitos para uma representação, enquanto um número real pode ser composto por infinitos dígitos. Portanto, a representação numérica tem como principais propriedades a serem consideradas sua granularidade e seu intervalo de representação. Devido a diferenças entre tamanho da palavra e do hardware e entre atraso de propagação e precisão, vários tipos de representação numérica foram propostos e utilizados.

Normalmente, em aritmética computacional, é utilizado como sistema numérico uma notação não redundante, posicional e por peso. Em um sistema não redundante cada número tem uma representação única, ou seja, não há duas seqüências numéricas que representem o mesmo valor. Na notação numérica posicional o valor de cada algarismo é determinado pelo seu símbolo e pela sua posição dentro do número. Um sistema numérico é por peso quando um dígito tem um peso diferente para cada posição. No sistema convencional, o peso de cada posição *i* é a *i-ésima* potência de um valor fixo B, que é a base da representação. As bases mais comuns neste sistema são a binária (base 2), a decimal (base 10), a octal (base 8) e a hexadecimal (base 16).

Aqui serão abordados diferentes sistemas de representação numérica, o que inclui o sistema normalmente usado, chamado neste trabalho de convencional. Alguns sistemas alternativos serão apresentados a título de comparação com o sistema numérico por resíduos (RNS), que é o principal foco deste estudo.

2.1 Sistema Convencional

Um número N é representado em um sistema convencional de base B por uma sequência de n dígitos $(d_{n-1}d_{n-2}...d_1d_0)$ onde $0 \quad i \quad n - 1$, d_i é um dígito e $d_i \quad \{0; 1;...;B-1\}$. A base do número representado pode ser indicada em subscrito. Por ser um sistema posicional, o valor é influenciado pela posição de cada dígito. (WEBER, 2004) Ou seja, o valor de N pode ser representado como

$$N = d_{n-1} \times w_{n-1} + d_{n-2} \times w_{n-2} + ... + d_1 \times w_1 + d_0 \times w_0$$

onde w_n é o peso referente à posição n. Esta expressão também pode ser representada pela seguinte fórmula:

$$\sum_{i=0}^{n-1} (d_i \times w_i)$$

Nos sitemas digitais atuais, utilizamos o sistema binário, onde a base B é 2 (dois) e $d_i \quad \{0; 1\}$. Isto se deve ao fato de só conseguirmos armazenar dois valores distintos para cada bit, 0 (zero) ou 1 (um). Além da base binária, outra base bastante utilizada em

sistemas computacionais é a base hexadecimal, onde B = 16 e d_i {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}. Como nestes sistemas a capacidade de representação é sempre finita, esses dois fatores estarão inevitavelmente relacionados entre si. Ou seja, com *n* bits, será possível representar somente 2^n valores. Para se aumentar o intervalo de representação, deverá se diminuir a granularidade, e a recíproca também é válida.

Como os números são representados como polinômios, a transformação entre bases pode ser feita calculando-se o número como um polinômio utilizando-se a base de destino. A Tabela 2.1 mostra alguns valores em diferentes bases e sua conversão para a base decimal, onde B = 10 e d_i {0,1,2,3,4,5,6,7,8,9}.

Tabela 2.1: Representação de valores sem sinal em diferentes bases.

Número	Valor em Decimal
1101_2	$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$
1312_4	$1 \times 4^3 + 3 \times 4^2 + 1 \times 4^1 + 1 \times 4^0 = 117$
57_8	$5 \times 8^1 + 7 \times 8^0 = 47$
$4A_{16}$	$4 \times 16^1 + 10 \times 16^0 = 26$

2.2 Sistema *Signed-Digit*

Um número no sistema *signed-digit* (SD) X é representado por

$X = (x_{n-1}...x_0.x_{-1}...x_{-k})$

onde x_i pode ser qualquer valor de − , ...,−1, 0, 1, ..., , ou seja, − x^i . No sistema numérico *signed-digit* se o valor de cada dígito estiver no intervalo 0 d_i B−1 a representação não é redundante. Quando cada dígito puder apresentar tanto valores positivos quanto negativos a representação é redundante. Os sistemas de números redundantes se destacam como sistemas numéricos em que *B* é menor do que o número de elementos no intervalo de representação de um dígito, isto é, B < − +1. Dessa forma, a redundância pode ser definida como a capacidade de representar um mesmo valor de várias formas distintas. O índice de redundância pode ser calculado pela fórmula:

= + + 1 − B

e para redundância mínima, $\lceil \frac{B-1}{2} \rceil \leq \alpha \leq B-1$.

Para uma representação simétrica, utiliza-se = .

2.2.1 Conversão de notação convencional para *signed-digit*

A conversão de um número *X* em notação convencional para um número *Y* em *signed-digit* é feita convertendo cada dígito x_i em um dígito de transferência b_{i+1} e um dígito reescrito d_i, sendo que

$x_i = b_{i+1} \times B + d_i$.

Esta conversão pode ser realizada através das seguintes fórmulas:

$$y_i = d_i + b_i$$

$$b_{i+1} = \begin{cases} 0 & \text{if } x_i < \alpha \\ 1 & \text{if } x_i \geq \alpha \end{cases}$$

A figura 2.1 mostra um exemplo de conversão de um número da notação tradicional para a *signed-digit*, com B = 4 e = = 2.

i	**6**	**5**	**4**	**3**	**2**	**1**	**0**
X_i		3	1	2	0	2	3
b_i(dígito de transferência)	1	0	1	0	1	1	
d_i(dígito reescrito)		$\overline{1}$	1	$\overline{2}$	0	$\overline{2}$	$\overline{1}$
$Y_i = b_i + d_i$	1	$\overline{1}$	2	$\overline{2}$	1	$\overline{1}$	$\overline{1}$

Figura 2.1: Conversão de um número para signed-digit

2.2.2 Conversão de notação *signed-digit* para convencional

A conversão de um número *signed-digit* para a notação convencional é bastante simples, exigindo apenas uma operação de subtração convencional. Um número *signed-digit Y* deve ser convertido para dois valores, um formado pelos seus dígitos positivos e com os demais substituídos pelo dígito 0 (zero); e o outro pelos dígitos negativos e tendo os demais substituídos por 0 (zero). Então é feita uma subtração do primeiro valor pelo segundo. Exemplo: Para um valor $Y = (1!45!2)_{10}$, X = 1050 − 0402 = 0648.

2.3 Sistema Logarítmico

O sistema numérico logarítmico pode simplificar multiplicações, divisões e potências. Entretanto, operações de soma e subtração tornam-se mais complexas. As conversões são um ponto crítico, pois podem ter custo elevado, e muitas vezes utilizam aproximações, o que pode levar a uma falta de precisão no resultado. Portanto, este sistema só poderá se mostrar vantajoso em sistemas que realizem muitas operações de multiplicação e divisão para poucas conversões, como por exemplo, em filtros digitais de tempo real. (KOREN, 2002)

2.3.1 Logaritmos

O logaritmo *x* de um número *y* em relação à base *b*, representado como $x = \log_b y$, é o expoente ao qual é necessário elevar *b* para se obter *y*. Ou seja, Um logaritmo *x* para a base *b* e o número *y* é definido como a função inversa de *b* na potência *y* ou b^y para quaisquer valores de *b* e *y*.

$$x = \log_b y \qquad b^x = y$$

$$x = \log_b(b^x)$$

Os logaritmos em relação à base b=10 também são chamados de logaritmos comuns, enquanto logaritmos em relação à base b=e=2,718... são chamados de logaritmos naturais.

A Figura 2.2 mostra o gráfico da função logarítmica para as bases 2, e e 10. Como podemos observar no gráfico, esta função tem uma singularidade em x=0; neste ponto a função tem o valor 1 para qualquer base.

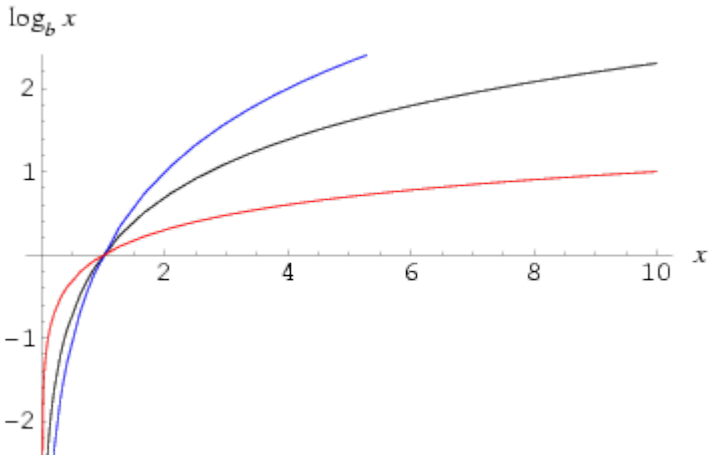

Figura 2.2 : Gráfico de logaritmos

2.3.1.1 Identidades Algébricas

Os logaritmos têm como identidades triviais as seguintes:

$$\log_b 1 = 0$$

$$\log_b b = 1$$

As principais propriedades algébricas dos logarítmos, que são usadas para facilitar cálculos, são mostradas nas fórmulas abaixo:

$$\log_b(M \times N) = \log_b M + \log_b N$$

$$\log_b \frac{M}{N} = \log_b M - \log_b N$$

$$\log_b N^y = y \times \log_b N$$

$$\log_b \sqrt[y]{x} = \frac{\log_b x}{y}$$

2.3.1.2 Mudança de Base

A mudança da base do logaritmo pode ser realizada facilmente aplicando-se a seguinte fórmula:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

2.3.1.3 Representação Numérica

Dado um número sem sinal X, obtemos o logarítmo de X na base r, e o denominamos L_x, ou seja, $L_x = \log_r X$. Assumindo que L_x é representado em binário com

n bits para a parte inteira e *k* bits para a parte fracionária, temos que *X* é representado no sistema numérico logarítmico (LNS) no seguinte formato:

$$L_x = x_{n-1}x_{n-2} \ldots x_0.x_{-1}x_{-2} \ldots x_{-k}.$$

A abrangência de representação no sistema logarítmico não inclui o valor zero, visto que r^{Lx} 0 para qualquer valor de L_x. Com *n* bits para a parte inteira e *k* bits para a parte fracionária temos que

$$-2^{n-1} \quad L_x \quad 2^{n-1} - 2^{-k}.$$

2.4 Sistema RNS

No sistema numérico por resíduos (RNS), representamos um valor por uma seqüência de valores menores para uma respectiva base. Uma base RNS é formada por um conjunto de números primos entre si, representada como $(m_k-1|m_{k-2}|...|m_1|m_0)$. Dois valores tomados dois a dois são primos entre si se, e somente se, seu MDC = 1. Cada elemento m_i da base é denominado um *modulus* enquanto que um conjunto de *modulus* $(m_k-1|m_{k-2}|...|m_1|m_0)$ é denominado *moduli*. Para calcular o valor em RNS, deve-se utilizar a operação MOD, que é o resto da divisão inteira, no valor a ser convertido para cada um dos elementos da base.

Para converter, por exemplo, o valor 27 em decimal para a base (7 | 5 | 3), temos que:

27 MOD 7 = 6
27 MOD 5 = 2
27 MOD 3 = 0

Portanto, este valor convertido pra RNS é $(6 \mid 2 \mid 0)_{RNS(7|5|3)}$

A Tabela 2.2 mostra alguns exemplos de valores em binário, decimal e a conversão para RNS.

Tabela 2.2: Valores no sistema convencional nas bases binária e decimal e o equivalente em RNS nas bases (7|5|3) e (7|5|4).

Binário	Decimal	RNS(7|5|3)	RNS (7|5|4)
0	0	(0 | 0 | 0)	(0 | 0 | 0)
1	1	(1 | 1 | 1)	(1 | 1 | 1)
10	2	(2 | 2 | 2)	(2 | 2 | 2)
11	3	(3 | 3 | 0)	(3 | 3 | 3)
100	4	(4 | 4 | 1)	(4 | 4 | 0)
101	5	(2 | 0 | 2)	(2 | 0 | 1)
110	6	(6 | 1 | 0)	(6 | 1 | 2)
111	7	(0 | 2 | 1)	(0 | 2 | 3)
1000	8	(1 | 3 | 2)	(1 | 3 | 0)
1001	9	(2 | 4 | 0)	(2 | 4 | 1)
1010	10	(3 | 0 | 1)	(3 | 0 | 2)
1011	11	(4 | 1 | 2)	(4 | 1 | 3)
1100	12	(5 | 2 | 0)	(5 | 2 | 0)
1101	13	(6 | 3 | 1)	(6 | 3 | 1)
1110	14	(0 | 4 | 2)	(0 | 4 | 2)
1111	15	(1 | 0 | 0)	(1 | 0 | 3)

A abrangência (ou alcance) *M* da representação RNS é obtida pela multiplicação de todos os elementos da base. Portanto, para a base RNS ($m_k-1|m_{k-2}|...|m_1|m_0$), M = $m_{k-1}\times m_{k-2}\times...\times m_1\times m_0$. Para a base (7|5|3), por exemplo, M = 7×5×3 = 105. Portanto, é possível representar 105 valores diferentes com a base (7|5|3). A base deve ser escolhida para satisfazer a abrangência desejada. Para qualquer valor *X*, sendo 0 X < M−1, há uma correspondência de um-para-um entre o valor de *X* e a representação RNS.

A abrangência de uma representação em RNS, normalmente obtida como um intervalo de 0 (zero) a M-1, pode ser mapeada para qualquer intervalo de *M* números inteiros consecutivos. Números negativos podem ser representados, por exemplo, fazendo a primeira metade do alcance representar os números negativos e a segunda metade representar o zero e os valores positivos de forma crescente.

Em um sistema digital, cada resíduo precisa de poucos bits quando comparados ao tamanho original da palavra em binário. Por exemplo, na base (7|5|3) o maior resíduo é representado com 3 bits, enquanto que para representar o alcance 105 em binário precisamos de 7 bits. Com os valores dos resíduos pequenos, as operações ficam simplificadas, diminuído a propagação de *carry*. No entanto a quantidade de bits para representar todos os resíduos respectivos a uma base RNS é maior do que o tamanho da palavra em binário. Para a base (7|5|3) precisamos de 8 bits para a representação RNS (3 bits para o resíduo de 7; 3 para o de 5; e 2 para o de 3), diminuindo assim, a eficiência da representação, pois estamos utilizando mais bits do que o necessário para uma representação binária.

2.4.1 Escolha da Base

A escolha da base RNS, ou *moduli*, é muito importante, pois afeta a eficiência da representação e a complexidade dos algoritmos aritméticos. Em geral, tenta-se pegar os menores *moduli* possíveis, visto que o desempenho das operações é determinado pelo pior desempenho dos *moduli*. Além disso, a magnitude precisa ser adequada para a representação desejada. Uma primeira abordagem é pegar números primos até obter a abrangência (ou alcance) necessária. Se este alcance for muito superior ao desejado, pode-se retirar algum número maior. Números muito pequenos podem ser multiplicados entre si, desde todos os *moduli* se mantenham sempre primos entre si. Também se pode utilizar potências destes números, o que se torna muito útil para potências de 2, pois facilita bastante a operação de MOD na representação binária.

Os estudos feitos em (WANG; SWAMY; AHMAD, 2003), (WANG et al., 2003) e (BI; JONES, 1988) mostram como o *moduli* para a representação em RNS pode ser escolhido de forma a minimizar o impacto das etapas adicionais necessárias para o uso de RNS. Como os módulos mais fáceis de se calcularem em sistemas digitais são as potências de dois, estes são bastante explorados para o uso em bases RNS. Em (WANG et al., 2003) são analisados quatro formatos de base e o impacto desta escolha na otimização e implementação de codificadores e decodificadores. Os formatos analisados são $\{2^n, 2^n+1, 2^n-1\}$, $\{2n, 2n+1, 2n-1\}$, $\{2^n, 2^n-1, 2^{n-1}-1\}$ e $\{2^{2n}+1, 2^n+1, 2^n-1\}$. Na Tabela 2.3 podemos ver os valores que poderiam ser utilizados para a base em cada um destes formatos para que a representação tenha abrangência equivalente ao sistema binário com 8, 16, 32 e 64 bits de tamanho.

Tabela 2.3:Bases RNS e suas abrangências (WANG et al., 2003).

Formato da Base	8 bits	16 bits	32 bits	64 bits
$\{2^n, 2^n+1, 2^n-1\}$	{8, 9, 7}	{64, 65, 63}	{2048, 2049, 2047}	$\{2^{22}, 2^{22}+1, 2^{22}-1\}$
{2n, 2n+1, 2n−1}	{8, 9, 7}	{42, 43, 41}	{1626, 1627, 1625}	{2642246, 2642247, 2642245}
$\{2^n, 2^n-1, 2^{n-1}-1\}$	{16, 15, 7}	{64, 63, 31}	{4096, 4095, 2047}	$\{2^{22}, 2^{22}-1, 2^{21}-1\}$
$\{2^{2n}+1, 2^n+1, 2^n-1\}$	{65, 9, 7}	{1025, 33, 31}	{262145, 513, 511}	$\{2^{34}+1, 2^{17}+1, 2^{17}-1\}$

3 OPERADORES ARITMÉTICOS

Este capítulo apresenta alguns algoritmos e arquiteturas de operadores aritméticos para a representação binária sem sinal.

3.1 Somadores

Quando o valor de uma posição precisa ser aumentado em uma unidade, seu símbolo é substituído pelo de valor imediatamente superior. Quando uma posição é ocupada pelo símbolo de maior valor e esta deve ser aumentada em uma unidade, esta posição recebe o valor nulo e a posição imediatamente superior deve ter seu valor aumentado em uma unidade. Para a soma de dois valores, os dígitos de cada posição devem ser somados. Quando o valor da soma de uma determinada posição ultrapassar o máximo permitido pela base, a posição seguinte deve receber uma sinalização de que precisa ser aumentada. Este sinal é chamado de "vai-um"ou *carry-out* (c_{out}). Quando um sinal é recebido por uma determinada posição para ser somado junto com os respectivos dígitos, este é chamado de "vem-um"ou *carry-in* (c_{in}).

Uma soma pode ser efetuada através de somadores para cada posição dos dígitos. Quando um destes somadores recebe apenas os dois dígitos da soma, ele normalmente é chamado de meio-somador ou *half-adder* (HA). Quando o sinal de *carry-in* também é considerado, o somador é chamado de somador completo ou *full-adder* (FA). Ambos são blocos bastante versáteis utilizados na síntese de somadores e diversos outros circuitos aritméticos. As saídas do somador completo binário com as entradas *x* e *y* são o bit de soma *s* e o bit de *carry-out* c_{out}, cujas fórmulas são as seguintes:

$s = x \quad y \quad c_{in}$

$s = x * y * c_{in} + !x* !y * c_{in} + !x * y * \ c_{in} + x * !y * !c_{in}$

$c_{out} = x*y + x*c_{in} + y*c_{in}$

3.1.1 Somador Ripple Carry

Os somadores do tipo *ripple carry* (RCA) são circuitos aritméticos simples de serem projetados, pois consistem em diversos somadores de um bit (*full-adders*) em série que somam os operandos e o *carry-in* (*carry-out* do estágio anterior). A Figura 3.1 mostra um somador RCA para 4 bits.

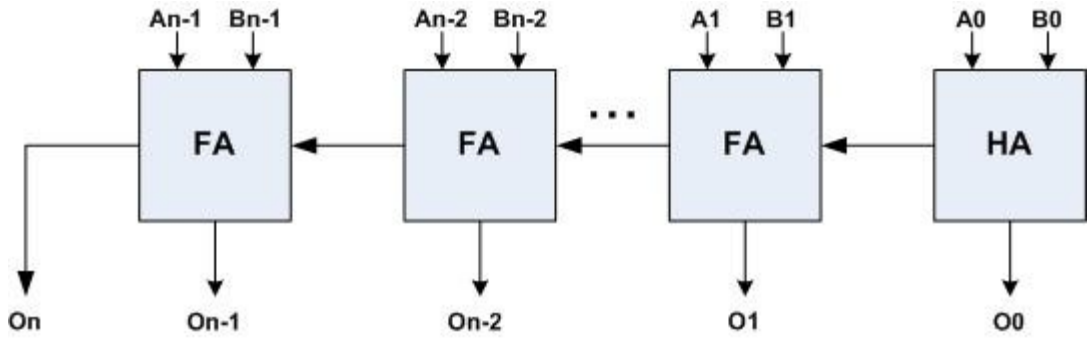

Figura 3.3 : Somador Ripple-carry.

Como os *full-adders* estão encadeados em seqüência, o pior caso do atraso ocorre quando o sinal de *carry* é gerado no primeiro estágio e propagado através de todos os estágios até o último. Este atraso é linearmente proporcional ao tamanho do somador, o que pode lhe tornar muito lento. Como temos um *half-adder* para cada posição dos vetores de entrada, a área também é linearmente proporcional ao tamanho das entradas do somador. Os custos de atraso e área de um somador *ripple carry* são mostrados nas fórmulas:

$T_{RCA} = n \times T_{FA}$

$A_{RCA} = n \times A_{FA}$

3.1.2 Somador Carry Select

O somador *carry select* (CSA) baseia-se na divisão da cadeia de *carry*. Os bits mais significativos (msb) e os menos significativos (lsb) são computados em paralelo. Como o *carry-in* dos bits msb só é conhecido após o processamento dos bits lsb, o subcircuito que calcula o resultado para os bits msb é duplicado, sendo que em um deles a entrada *carry-in* é o valor 0 (zero) e no outro é o valor 1(um). Para escolher o resultado correto é utilizado um multiplexador para cada bit msb da saída do circuito. Estes multiplexadores têm como entrada os valores calculados por cada um dos subcircuitos de soma dos bits msb. O sinal de *carry-out* da soma dos bits lsb é utilizado para a escolha do resultado correto.

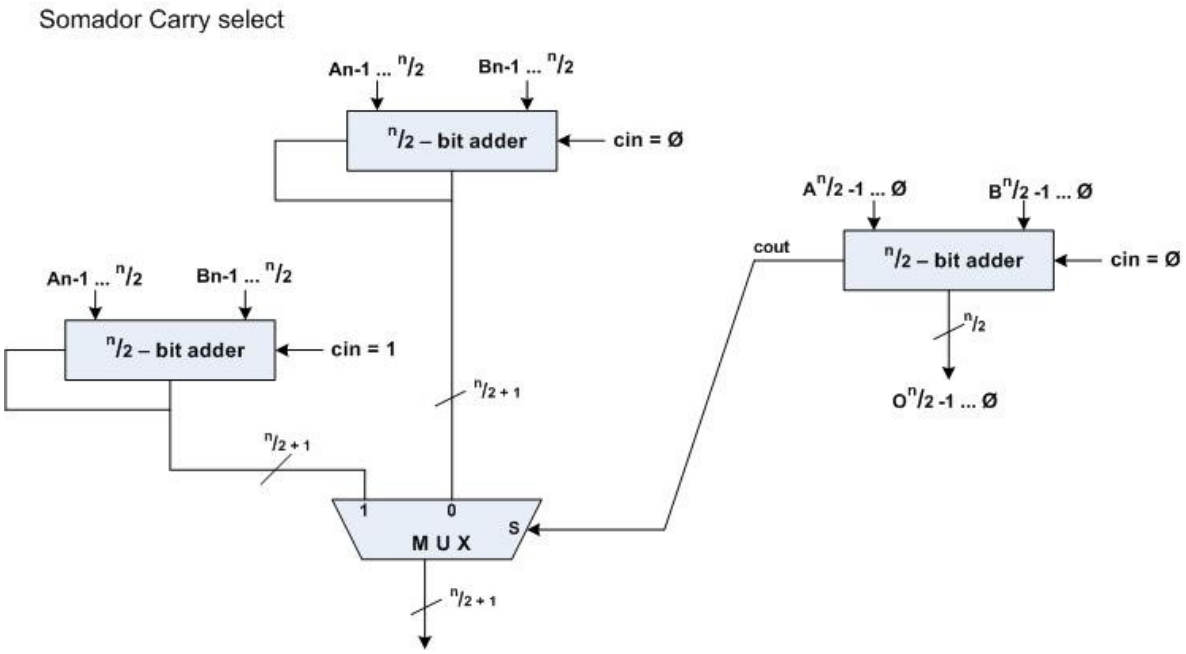

Figura 3.4: Somador *Carry Select*

Há um custo grande de área com esta técnica, pois há um acréscimo de 50% de *full-adders* e mais os multiplexadores. Mas seu tempo de atraso é diminuído, visto que ambas as cadeias de *carry* têm o mesmo atraso, que é equivalente a metade do atraso original, e o atraso de um multiplexador é pequeno, e dependendo do tamanho do circuito pode ser insignificante. Portanto, os custos de área e atraso desta arquitetura são determinados pelas fórmulas a seguir:

$T_{CSA} = T_{ADD(n/2)} + T_{MUX}$

$A_{CSA} = 3 \times A_{ADD(n/2)} + (n/2) * A_{MUX}$

Este método pode ser realçado se as correntes de *carry* continuarem a ser divididas até 4 ou 8 partes, aumentando a área e diminuindo o tempo de propagação. As correntes duplicadas de RCA também podem ser usadas em cascata, onde as divisões do circuito não necessitam ser no mesmo tamanho. Esta divisão é feita com a finalidade de que o atraso da corrente seguinte seja o mesmo da corrente anterior mais o atraso do multiplexador. Também é possível substituir estes subcircuitos de somadores por outra arquitetura mais eficiente que o RCA.

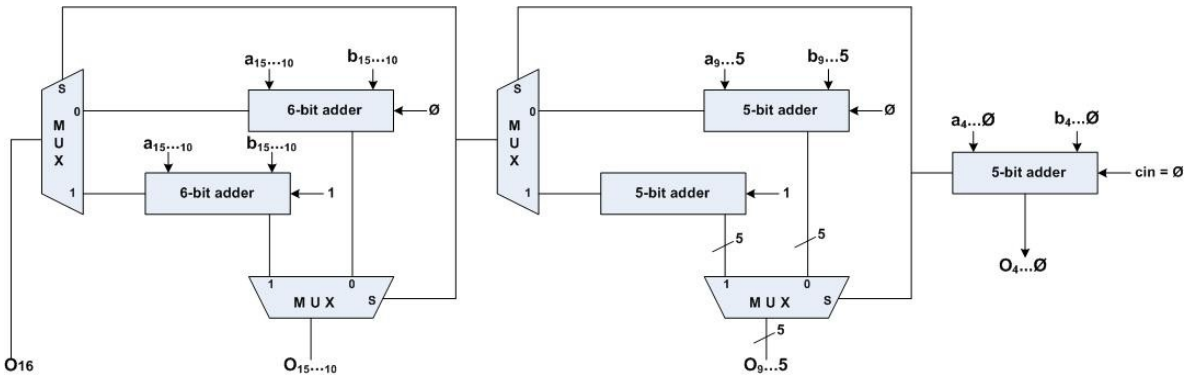

Figura 3.5: Somador *Carry Select* de 16 bits

3.1.3 Somador *Carry Skip*

O somador do tipo *carry skip*, assim como o CSA, tem sua estrutura baseada na divisão da cadeia de *carry*. Para cada divisão da cadeia da *carry*, é utilizado um circuito para antecipar o *carry-out* desta estrutura. Este detector de *carry* parte do princípio que se em cada posição das entradas ao menos um dos bits for 1, haverá um *carry*. Então, para um somador de *n* bits, utiliza-se n portas OR entre os bits da mesma posição nas entradas e uma porta AND de *n* entradas com todos estes resultados. É utilizada uma porta OR que tem como entradas a previsão de *carry* e o *carry* real do circuito, e a saída é ligada ao *carry-in* do circuito seguinte. Com esta estrutura, o sinal de *carry-out* é calculado de uma maneira mais rápida, pois se uma propagação de *carry-out* for detectada em alguma dessas cadeias de *carry*, o bit de *carry-in* já é setado na cadeia seguinte.

3.1.4 Somador *Carry Lookahead*

O somador do tipo *carry lookahead* (CLA) utiliza como princípio o cálculo de *carries* para otimizar o desempenho. Sabemos que só haverá *carry* em um FA quando ao menos duas de suas entradas tiverem o valor 1 (um). Aplicando este conceito recursivamente para obter os valores dos *carry-ins*, obtemos as seguintes fórmulas:

$Cout_0 = A_0*B_0 + A_0*Cin_0 + B_0*Cin_0$

$Cout_1 = A_1*B_1 + A_1*(A_0*B0 + A_0*Cin_0 + B_0*Cin_0) + B_1*(A_0*B_0 + A_0*Cin_0 + B_0*Cin_0)$

Entretanto, este método de cálculo de *carry* se torna impraticável, visto que normalmente são utilizados somadores com valores muito grandes na entrada, e estes circuitos ficariam muito grandes.

Devido a essas restrições, os somadores de *carry lookahead* normalmente têm seu funcionamento baseado nos sinais de "propagação" (quando um sinal de *carry* se propaga através do estágio) e "geração" (quando um sinal de *carry* é gerado, mesmo quando não há *carry-in* no estágio). É possível agrupar estes sinais em blocos e calcular, recursivamente, a propagação e a geração de *carry* em diferentes partes do circuito, não dependendo do estágio anterior.

3.1.5 Somadores de Prefixo Paralelos

Em um somador de prefixo paralelo (PPA), a soma pode ser funcionalmente divida em três etapas: pré-processamento, computação de prefixo e pós-processamento (LIU et al., 2003).

No pré-processamento os vetores de entrada são utilizados para calcular os sinais necessários às etapas posteriores. Para cada posição das entradas do somador, são calculados os valores de geração (G) e propagação (P) de *carry*. A geração do *carry* somente ocorre quando os dois sinais de entrada são 1 (um). E a propagação do *carry* ocorre quando um dos sinais de entrada é igual a 1 (um). Portanto, as fórmulas para esta etapa do processamento são:

$P_i = A_i$ XOR B_i

$G_i = A_i$ AND B_i

Como estas equações podem ser efetuadas em paralelo para todas as posições dos vetores de entrada, não acrescenta um grande atraso na computação, e o custo de área é proporcional ao tamanho da entrada.

$T_{PPA\text{-}PRE} = T_{XOR}$

$A_{PPA\text{-}PRE} = n * (A_{XOR} + A_{AND})$

No cálculo de prefixo os sinais obtidos do pré-processamento são processados por uma estrutura de prefixo, que irá calcular todos os sinais necessários à realização da soma na última etapa. Neste processamento, são calculados os valores dos sinais de geração e propagação de *carry* entre posições adjacentes. Para o cálculo dos novos sinais, são utilizadas as seguintes fórmulas:

P = P' AND P''

G = (G' AND P'') OR G''

A etapa de processamento se encerra quando já foi calculado os valores de propagação e geração de *carry* desde a posição 0 (zero) até cada uma das outras posições.

E por fim, no pós-processamento, o vetor com o resultado da soma e o *carry* de saída são calculados a partir dos sinais provenientes das etapas anteriores.

Os dois modelos mais usados para a composição de PPAs foram propostos por Brent-Kung e por Kogge-Stone, mostrados na figura abaixo. O primeiro minimiza o número das computações, com menos área e um desempenho mais lento, com seu atraso determinado por *2*log(n-2)*. O segundo apresenta a execução a mais rápida possível de somadores paralelos do prefixo, com o seu atraso determinado pelo *log(n)*, mas com um grande consumo de área. (PARHAMI, 2000) Também são feitos projetos híbridos, que equilibram a relação entre área consumida e atraso do circuito.

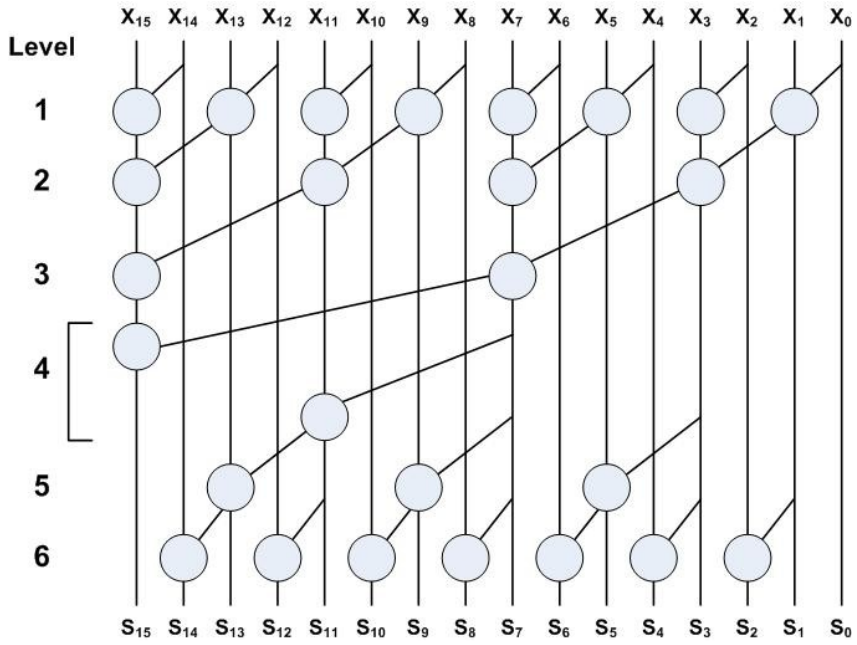

Figura 3.6: Somador PPA Brent-Kung

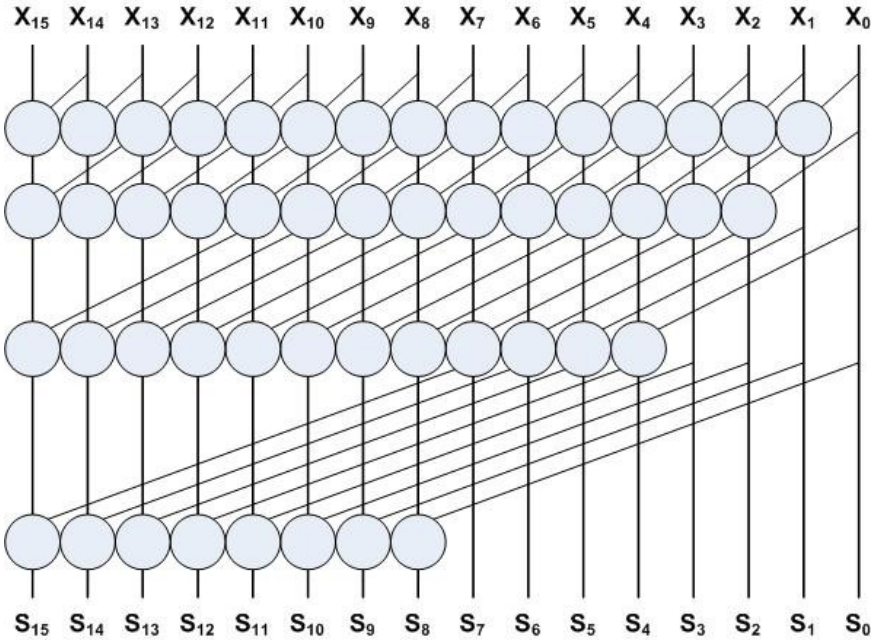

Figura 3.7: Somador PPA Kogge-Stone

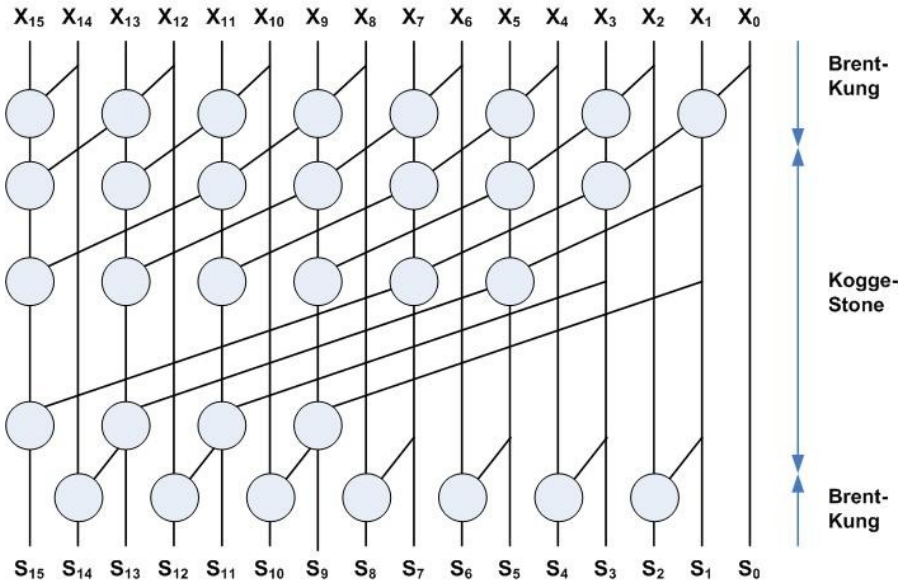

Figura 3.8: Somador PPA híbrido

3.2 Multiplicadores

Neste capítulo serão abordados os algoritmos de multiplicação para a representação numérica convencional, sem sinal, na base binária. As arquiteturas de multiplicadores se dividem basicamente em duas categorias, a multiplicação seqüencial e a multiplicação paralela.

O algoritmo de multiplicação segue a metodologia normalmente utilizada para multiplicações feitas manualmente no papel. Este método consiste em duas etapas. Na primeira os dígitos do multiplicador são analisados individualmente, gerando um produto parcial para cada um destes dígitos multiplicados pelo multiplicando. Na

segunda etapa os produtos parciais são somados para a geração do resultado final. Na Tabela 3.1 é mostrada a tabela verdade da multiplicação de dois dígitos para a base binária. Como podemos observar, esta operação é equivalente à função E (AND) dos dois sinais de entrada. A Figura 3.7 mostra um exemplo de multiplicação nesta base.

Tabela 3.4: Multiplicação na base binária

x_i	y_i	$p_i = x_i$ y_i
0	0	0
0	1	0
1	0	0
1	1	1

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 1 \\
\times & & & & 1 & 0 & 1 \\
\hline
 & & & & 1 & 1 & 1 \\
+ & & & 0 & 0 & 0 & \\
+ & & 1 & 1 & 1 & & \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

Figura 3.9: Exemplo de multiplicação de números sem sinal na base binária

3.2.1 Multiplicação Seqüencial

A multiplicação seqüencial é baseada em operações soma-deslocamento, de forma semelhante à multiplicação feita manualmente no papel. São obtidos produtos parciais para cada bit do multiplicador. Estes resultados parciais são acumulados para a obtenção do produto. Como cada bit examinado é uma posição mais alta que a anterior, cada produto parcial deve ser deslocado para a esquerda um bit a mais que o produto parcial anterior antes de ter o seu valor acumulado.

3.2.1.1 Multiplicação por Deslocamento

A forma mais simples de multiplicação é através do deslocamento dos bits, com o acréscimo do símbolo de valor nulo na posição de menor peso. A cada deslocamento para a esquerda, um valor é multiplicado pelo valor da base B, o que equivale, nos sistemas que utilizam a notação binária, à multiplicação por 2. O deslocamento pode ser aplicado diversas vezes, permitindo a multiplicação por qualquer potência da base.

3.2.1.2 Multiplicação Soma-Deslocamento

Na multiplicação soma-deslocamento, os bits do multiplicador são analisados individualmente, gerando um produto parcial para cada um destes bits multiplicados pelo multiplicando. Os produtos parciais são somados para a geração do resultado final. Como cada produto parcial é gerado pela multiplicação de bits do multiplicador de diferentes posições, os produtos parciais são deslocados para a posição correspondente.

Ao invés de se esperar a geração de todos os produtos parciais e somá-los no final da operação, pode-se somar cada produto parcial a uma soma acumulada imediatamente após este produto ser gerado. Como cada produto parcial é sempre uma posição maior que o anterior, cada produto parcial a ser adicionado deve ser deslocado uma posição à esquerda a mais que o produto anterior; ou pode-se manter o produto parcial e deslocar a soma acumulada para a direita, obtendo-se uma operação equivalente.

3.2.1.3 Somadores Parciais

A multiplicação por somadores parciais é semelhante à convencional, mas os resultados parciais são somados por diversas instâncias de somadores.

3.2.1.4 Booth

Os esquemas de multiplicação indireta têm um atraso proporcional a *n* ciclos, onde *n* é o tamanho dos operandos de entrada. Para tornar a multiplicação mais rápida, a abordagem de varredura de múltiplos bits é adotada, possibilitando a redução dos produtos parciais conseqüentemente diminuindo o atraso.

3.2.2 Multiplicação Paralela

Na multiplicação paralela, as diferentes interações de adição podem ser realizadas por diferentes replicações do circuito de um estágio, em vez de usar sempre o mesmo circuito. Com isso, as operações podem ser sobrepostas e o tempo para completar a multiplicação pode ser reduzido. Existem várias arquiteturas de multiplicadores paralelos propostas, entre elas estão as árvores de Wallace e os *arrays* de multiplicadores.

3.2.2.1 Árvores de Wallace

Na operação de multiplicação, a soma dos produtos parciais é o processo que mais precisa de tempo, portanto são estudadas otimizações para estas somas. Uma árvore de Wallace é um somador *bit-slice* que soma bits da mesma posição. Árvore de Wallace W_3 de 3 bits é na verdade um somador completo, pois recebe com entrada três bits de uma posição i e seu resultado é um bit da posição i e um bit da posição *i+1*. Árvores de Wallace de mais bits podem ser construídas a partir de árvores de Wallace menores, conforme pode ser visto na Figura 5.3. Uma árvore de Wallace de 7 bits W_7 pode ser construída a partir de quatro W_3; e uma de 15 bits pode ser construída a partir de duas W_7 e três W_3. Para *k* entradas, uma árvore de Wallace tem $\log_2(k+1)$ saídas.

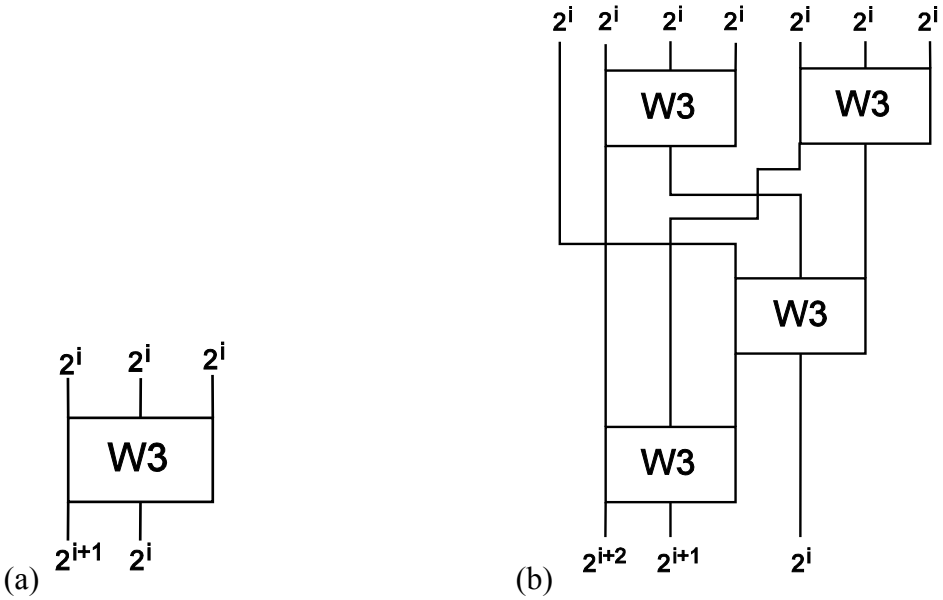

Figura 3.10: Árvores de Wallace de (a) 3 bits e (b) 7 bits

A Tabela 3.2 mostra combinações tradicionais entre árvores de Wallace e seus atrasos.

Tabela 3.5: Combinações e atrasos de árvores de Wallace

k	Combinações W_i	$_T$
3	W_3	4_{FA}
6	$4 \times W_3$	$3 \times 4_{FA}$
7	$(4 \times W_3)$ ou (W_7)	$3 \times 4_{FA}$
15	$(11 \times W_3)$ ou $(3 \times W_3 + 2 \times W_7)$	$5 \times 4_{FA}$

3.2.2.2 Array de Multiplicação

Sendo A e X números inteiros sem sinal de tamanhos m e n, temos P de tamanho $m+n$ como produto de A e X. Esta multiplicação $P = A \times X$ é chamada de $m \times n$. Para se realizar esta multiplicação, deve-se seguir dois passos: a geração dos produtos parciais e a soma destes produtos parciais para a obtenção do produto final. Esta soma pode ser feita por um *array* de *full-adders*, conforme ilustrado na Figura 3.9. Os *carries* gerados em uma coluna devem participar da soma da coluna onde está sendo calculado o dígito imediatamente superior do produto. Para evitar uma cadeia muito grande de somadores, pode-se agrupá-los na última linha como um somador do tipo *ripple carry*.

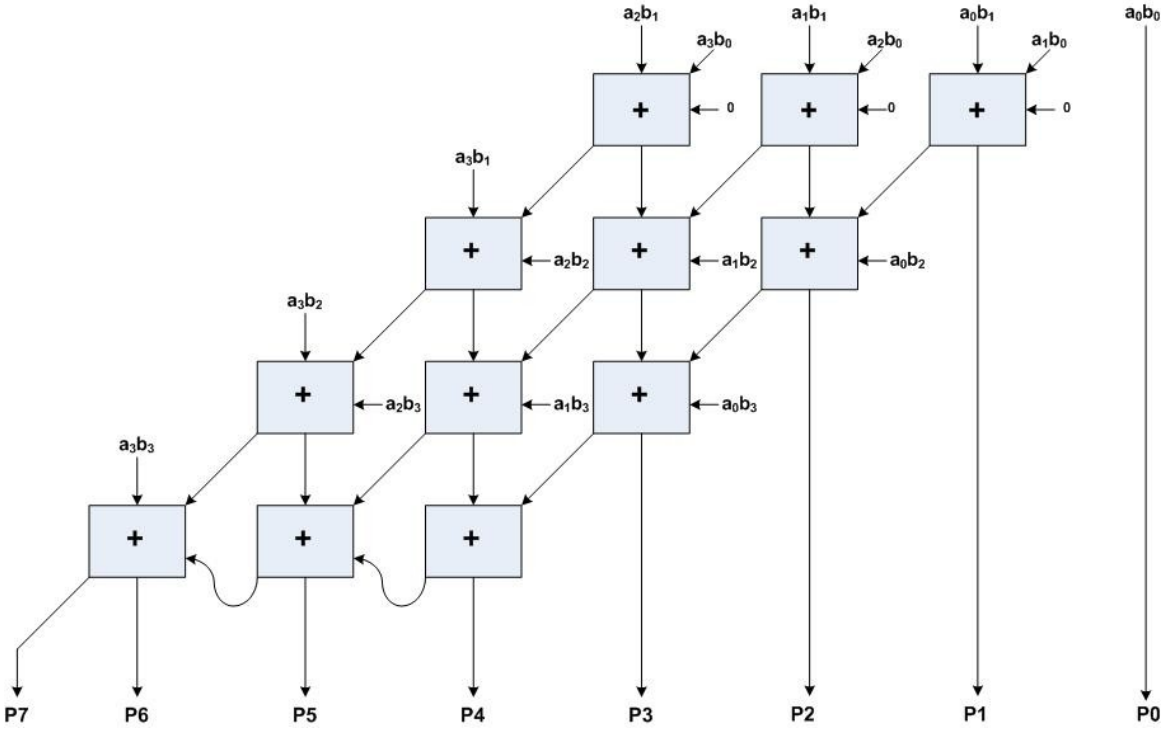

Figura 3.11: *Array* de Multiplicação

3.2.2.3 Multiplicação Modular

Uma estrutura modular pode ser utilizada para a geração e soma de produtos parciais de forma recursiva. Este método pode ser vantajoso para números com muitos dígitos. Módulos de multiplicação n×n podem ser transformados em um multiplicador 2n×2n. Considerando-se como denotador da função de concatenação. Para um multiplicando A de 2n bits, podemos particioná-lo na sua metade mais significativa A_H e na menos significativa A_L, cada uma delas com n bits, sendo que A = A_H A_L. Similarmente, o multiplicador X de 2n bits pode ser particionado em X_H e X_L de n bits cada, com X = X_H X_L. A multiplicação A×X de 2n×2n pode ser realizada por um conjunto de multiplicações n×n, cada uma gerando subprodutos de tamanho 2n, conforme ilustrado nas equações a seguir:

$$P = A \times X$$

$$P = A_H \; A_L \times X_H \; X_L$$

$$P = (A_H \times X_H + A_L \times X_H) + (A_H \times X_L + A_L \times X_L)$$

O alinhamento dos produtos parciais a serem somados é bastante importante, e pode ser visto na Figura 3.10. Como A_H é *n* posições de bit mais elevada que A_L, o produto $A_H \times X_L$ é *n* posições de bit mais superiores ao produto $A_L \times X_L$. O mesmo cuidado deve-se ter com o a diferença de alinhamento entre X_H e X_L. Este método pode ser utilizado recursivamente, possibilitando a formação de multiplicadores 16 × 16 como o da Figura 3.11 ou estruturas ainda maiores.

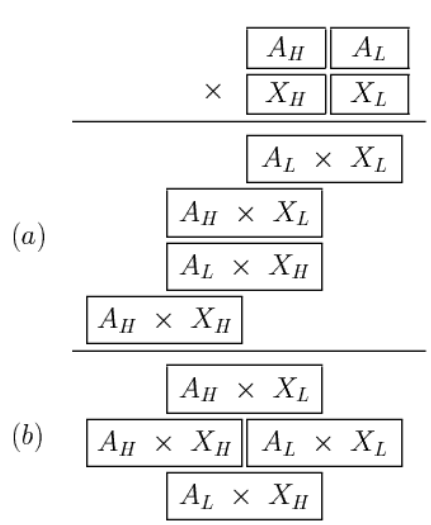

Figura 3.12: Alinhamento de quatro subprodutos nas formas (a) direta e (b) organizadas na estrutura modular(KOREN, 2002).

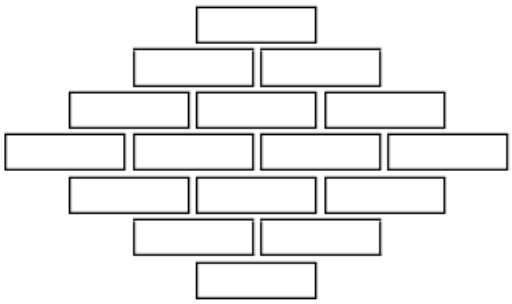

Figura 3.13: Alinhamento de 16 produtos parciais (KOREN, 2002).

3.2.2.4 Multiplicação Modular Aditiva

A multiplicação modular aditiva, assim como a modular, se baseia no uso recursivo de multiplicadores maiores para a construção do circuito. Porém, diferentemente da multiplicação modular, onde a soma dos mintermos é feita separadamente, na multiplicação modular aditiva os valores já são multiplicados e somados em cada bloco.

A estrutura de um multiplicador modular aditivo de 4 bits é semelhante ao de um multiplicador *array*, mas também realiza a soma de dois valores de 4 bits. Para isto, são utilizadas entradas de FAs que antes recebiam o valor zero e são adicionados também 4 FAs, conforme a Figura abaixo.

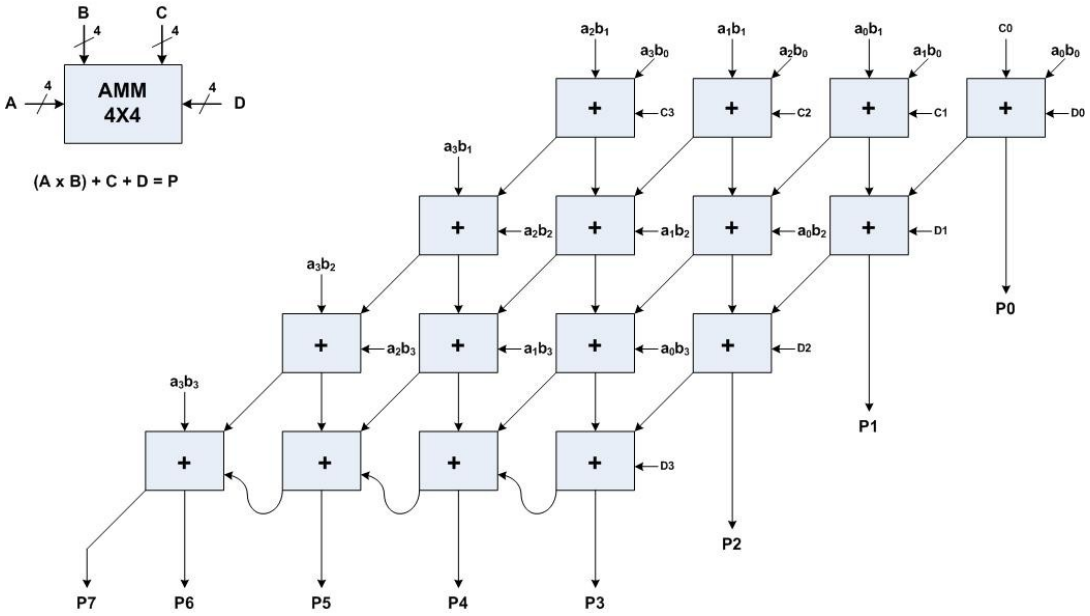

Figura 3.14: Multiplicador Modular aditivo de 4 bits.

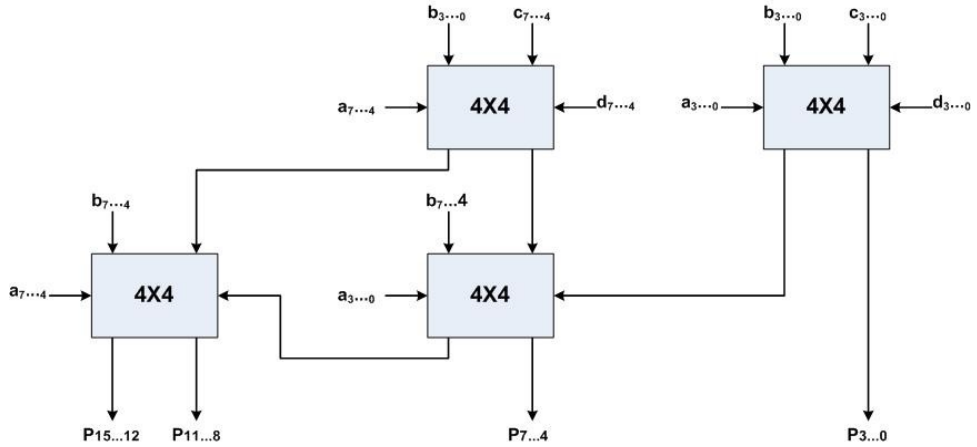

Figura 3.15: Multiplicador Modular Aditivo de 8 bits.

3.2.3 Multiplicação Seqüencial X Paralela

Como foi visto, os circuitos multiplicadores podem ser implementados em dois tipos diferentes de circuitos: seqüenciais ou paralelos. Os multiplicadores seqüenciais realizam as operações através de diversas iterações no mesmo componente, utilizando valores acumulados que são atualizados a cada ciclo do relógio. Este tipo de circuito, apesar de ocupar uma menor área, têm uma maior demora na resposta, pois somente após o cálculo de um determinado valor que o valor seguinte pode começar a ser computado. Já nos multiplicadores paralelos, a área é maior, visto que os mesmos componentes são instanciados várias vezes para que todos os valores possam ser calculados. Entretanto, estes têm um tempo médio de reposta menor, visto que enquanto os dados são propagados pelo circuito, é possível já fornecer novos dados para os estágios iniciais.

Este trabalho vai focar nos circuitos paralelos, que serão implementados e comparados com os circuitos RNS.

4 OPERADORES RNS

As operações aritméticas em RNS podem ser feitas em cada resíduo de forma independente, evitando sinais de *carry* entre os diferentes resíduos nas operações de adição, subtração e multiplicação, o que diminui o tempo de propagação de *carry*. Como as operações são feitas independentemente em cada resíduo, estas podem ser feitas em paralelo, tornando as operações mais rápidas. Entretanto, para operações como divisão, teste de sinal, comparação de magnitude ou detecção de *overflow* (quando o resultado da operação ultrapassa o alcance da base) é necessário transformar os resíduos para uma representação intermediária ou utilizar algoritmos mais elaborados. Esta complexidade adicional limita o uso do RNS, pois torna o custo destas operações muito elevado. Por isso, RNS não se mostra vantajoso em processadores de propósito geral, apesar de ser promissor em algumas aplicações específicas que utilizam basicamente adição, subtração e multiplicação.

Uma operação de soma pode ser realizada somando-se os elementos de mesma posição, ou seja, relativos ao mesmo *modulus*, e depois fazendo a operação de MOD (resto da divisão inteira) para cada um deles. Para os valores (2 | 3 | 2) e (2 | 2 | 2) na base (7 | 5 | 3), equivalentes aos valores decimais 23 e 2, respectivamente, a soma ficaria:

Conversão dos valores de decimal para RNS:

$$23 = (2 \mid 3 \mid 2)_{RNS(7|5|3)}$$

$$2 = (2 \mid 2 \mid 2)_{RNS(7|5|3)}$$

Operação de soma:

$$23 + 2 = (2 \mid 3 \mid 2)_{RNS(7|5|3)} + (2 \mid 2 \mid 2)_{RNS(7|5|3)}$$

Para cada posição é feita a soma seguida da operação *mod*

$$(2 + 2)\text{MOD}(7) = (4)\text{MOD}(7) = 4$$

$$(3 + 2)\text{MOD}(5) = (5)\text{MOD}(5) = 0$$

$$(2 + 2)\text{MOD}(3) = (4)\text{MOD}(3) = 1$$

Os valores obtidos são agrupados formando o *moduli* da resposta:

$$(2 \mid 3 \mid 2)_{RNS(7|5|3)} + (2 \mid 2 \mid 2)_{RNS(7|5|3)} = (4 \mid 0 \mid 1)_{RNS(7|5|3)}$$

Conversão do valor para decimal:

$$(4 \mid 0 \mid 1)_{RNS(7|5|3)} = 25$$

Na realização de diversas operações aritméticas em seqüência, não há necessidade de se fazer conversões para o sistema convencional durante os cálculos. Pode-se fazer somente as conversões das entradas e da saída final. Com isso, pode-se simplificar a computação.

Operações aritméticas de subtração e multiplicação também podem ser realizadas de forma análoga a adição, realizando a operação desejada em cada um dos resíduos de forma independente e seguida da operação MOD. Utilizando o símbolo para representar o operador de adição, subtração ou multiplicação, temos que, para uma base (m_2, m_1, m_0):

$$X = (x_2 \mid x_1 \mid x_0)$$

$$Y = (y_2 \mid y_1 \mid y_0)$$

$$Z = X \quad Y$$

$$z_2 = (x_2 \quad y_2) \text{MOD}(m_2)$$

$$z_1 = (x_1 \quad y_1) \text{MOD}(m_1)$$

$$z_0 = (x_0 \quad y_0) \text{MOD}(m_0)$$

4.1 Codificação de notação convencional para RNS

Para converter um valor *X*, sendo 0 X < M, para RNS obtêm-se os restos (ou resíduos) da divisão inteira deste valor por cada um dos elementos mi da base. Como o resto *r* da divisão inteira de um valor *x* dividido por y é representado por r = x MOD y, para cada *modulus* em ($m_{k-1}|m_{k-2}|...|m_1|m_0$), temos que x_i = X MOD m_i, e o número *X* pode ser representado em RNS como $X = (x_{k-1}|x_{k-2}|\ldots|x_1|x_0)_{RNS(mk-1|mk-2|...|m1|m0)}$.

Por exemplo, para converter o número 23 para RNS utilizando a base (7|5|3) é feita a operação 23 MOD m_i, para cada elemento m_i da base, obtendo-se assim os respectivos resíduos:

$$(23)\text{MOD}(7) = 2$$
$$(23)\text{MOD}(5) = 3$$
$$(23)\text{MOD}(3) = 2$$

Portanto, 23 pode ser representado em RNS como $(2|3|2)_{RNS(7|5|3)}$.

O custo adicionado pelas etapas de codificação e decodificação tem sido a principal dificuldade encontrada para o uso de RNS. Entretanto, como podemos ver em (MOHAN, 2002), o estudo para a otimização dessas conversões têm sido bastante abordado recentemente. Os trabalhos feitos por (PREMKUMAR, 2002), (SOUDRIS et al., 2001) e (RE et al., 2001) focam na otimização de processos de codificação e decodificação RNS.

No trabalho de (BI; JONES, 1988) é demonstrada uma implementação eficiente de conversor RNS para uma base no formato $\{2^n + 1, 2^n, 2^n - 1\}$. Consideramos *X* como um número binário de tamanho *3n*, a ser convertido para o valor (x_1, x_2, x_3) em RNS. Para gerar o valor de x_2, basta utilizar somente os *n* bits mais significativos. Para obter os valores de x_1 e x_3 se utiliza o codificador mostrado na figura 4.1. Para gerar as entradas deste codificador o número *X* é decomposto nos números y_0, y_1 e y_2, de *n* bits cada, conforme demonstrado abaixo:

$$X = \underbrace{x_{N-1} \ldots x_{2n}}_{y_2} \underbrace{x_{2n-1} \ldots x_n}_{y_1} \underbrace{x_{n-1} \ldots x_0}_{y_0}$$

Como pode ser visto na figura 4.1, este codificador é formado por um somador de *n+1* bits e dois somadores/subtratores de *n* bits, e pode ser facilmente configurado para o cálculo de x_1 ou x_3, alterando-se apenas um sinal de entrada. Como estes circuitos manipulam dados independentes, eles podem operar em paralelo no processo de codificação, fazendo com que o atraso total da codificação seja equivalente a apenas um destes circuitos.

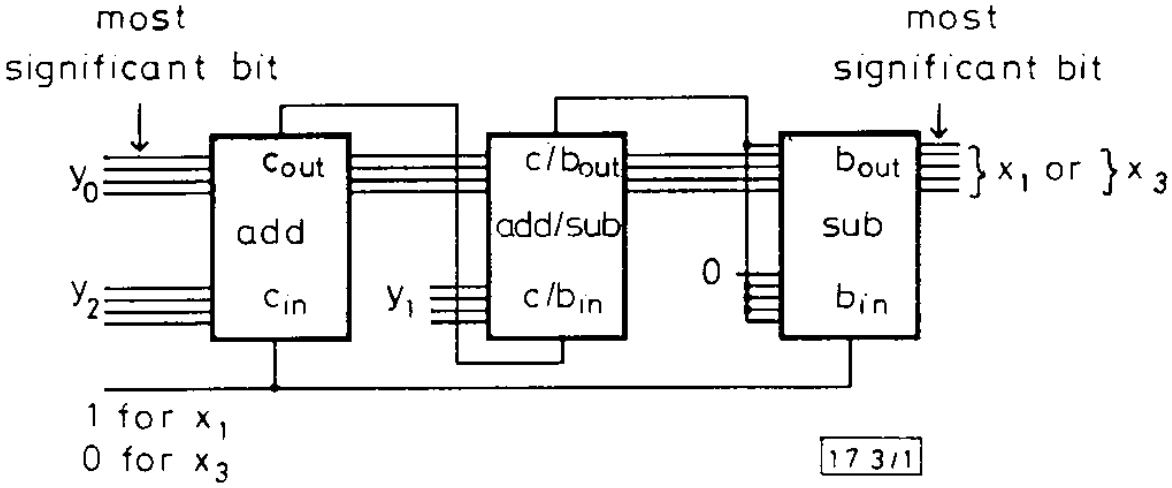

Figura 4.16: Codificador RNS (BI; JONES, 1988)

4.2 Decodificação de RNS para notação convencional

Para converter um valor de RNS para binário ou decimal aplica-se o Teorema Chinês do Resto (*Chinese Remainder Theorem* - CRT), cuja fórmula é:

$$X = (x_{k-1}|...|x_2|x_1|x_0)_{RNS} = \left(\sum_{i=0}^{k-1} M_i \times (\alpha_i \times x_i) MOD(m_i) \right) MOD(M)$$

Onde, por definição, $M_i = M/m_i$, e $\alpha_i = (M_i^{-1})MOD(m_i)$, que é o inverso multiplicativo de M_i em relaçao a m_i. O inverso multiplicativo de um valor x_i em relação a m_i é o número α_i tal que:

$(x_i \times \alpha_i)MOD(m_i) = 1$

Por exemplo, para converter $X = (2|3|2)_{RNS(7|5|3)}$ para decimal tem-se $M_2 = 105/7 = 15$; $M_1 = 105/5 = 21$; e $M_0 = 105/3 = 35$.

O inverso multiplicativo M_i em relação a cada elemento m_i da base é respectivamente $\alpha_2 = 1$, $\alpha_1 = 1$ e $\alpha_0 = 2$. Portanto tem-se que:

$X = ((M_2 \times \alpha_2 \times x_2) + (M_1 \times \alpha_1 \times x_1) + (M_0 \times \alpha_0 \times x_0))MOD(M)$

$X = ((15 \times 1 \times x_2) + (21 \times 1 \times x_1) + (35 \times 2 \times x_0))MOD(105)$

Portanto, para qualquer valor X, para a base RNS (7|5|3), temos que:

$X = ((15 \times x_2) + (21 \times x_1) + (70 \times x_0))MOD(105)$

E para o exemplo $X = (2|3|2)_{RNS(7|5|3)}$, obtemos

$X = ((15 \times 2) + (21 \times 3) + (70 \times 2))MOD(105)$

$X = (30 + 63 + 140)MOD(105)$

$X = (233)MOD(105) = 23$

Uma representação RNS também pode ser vista como uma soma ponderada. Por exemplo, $(2|3|2)_{RNS(7|5|3)}$ pode ser representado como:

$(2|3|2)_{RNS(7|5|3)} = (2|0|0)_{RNS(7|5|3)} + (0|3|0)_{RNS(7|5|3)} + (0|0|2)_{RNS(7|5|3)}$

$(2|3|2)_{RNS(7|5|3)} = 2 \times (1|0|0)_{RNS(7|5|3)} + 3 \times (0|1|0)_{RNS(7|5|3)} + 2 \times (0|0|1)_{RNS(7|5|3)}$

Para obtermos os valores de $(1|0|0)_{RNS(7|5|3)}$, $(0|1|0)_{RNS(7|5|3)}$ e $(0|0|1)_{RNS(7|5|3)}$, podemos aplicar o Teorema Chinês do Resto, onde temos $M_2 = 15$; $M_1 = 21$; $M_0 = 35$; $_2 = 1$; $_1 = 1$; $_0 = 2$. Os resíduos iguais a zero não contribuirão para a soma. Portanto tem-se que:

$(1|0|0)_{RNS(7|5|3)} = (15 \times 1 + 0 + 0)MOD(105) = 15$

$(0|1|0)_{RNS(7|5|3)} = (0 + 21 \times 1 + 0)MOD(105) = 21$

$(0|0|1)_{RNS(7|5|3)} = (0 + 0 + 35 \times 2)MOD(105) = 70$

Logo, podemos decodificar qualquer valor em RNS (7|5|3) fazendo:

$X = ((15 \times x2) + (21 \times x1) + (70 \times x0))MOD(105)$

Para uma base no formato $\{2^n + 1, 2^n, 2^n - 1\}$, conforme demonstrado em (BI; JONES, 1988), o circuito de decodificação de RNS para binário pode ser implementado conforme a Figura 3.2. O resultado da conversão é a concatenação dos valores y_0, y_1 e y_2 obtidos nesse circuito.

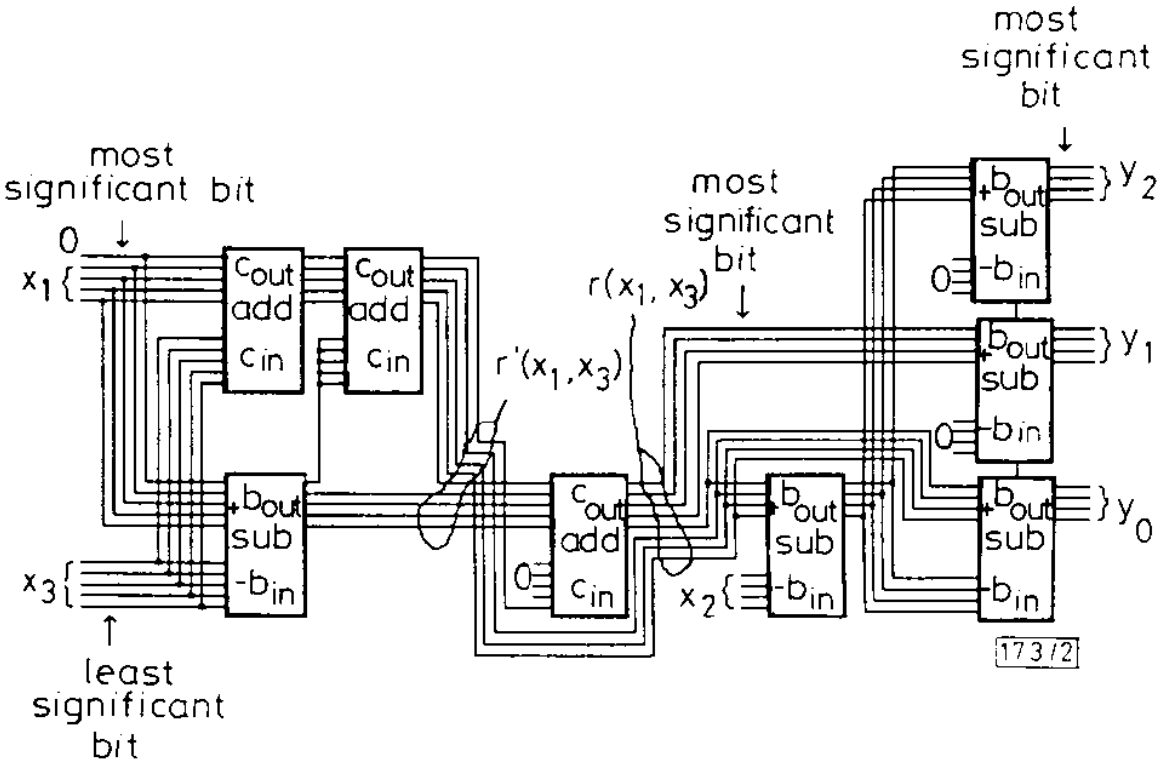

Figura 4.17: Decodificador RNS (BI; JONES, 1988)

5 IMPLEMENTAÇÃO E RESULTADOS

Para poder melhor analisar as características das diferentes arquiteturas de circuitos aritméticos, foram implementados alguns somadores e multiplicadores no sistema binário para 16, 32 e 64 bits. Também foram implementados circuitos equivalentes no sistema RNS, além dos codificadores de binário para RNS e de RNS para binário. Estas implementações foram feitas utilizando-se a linguagem de descrição de hardware Verilog.

5.1 Operadores Aritméticos

Os somadores implementados foram os seguintes: *ripple carry*, *carry look-ahead*, *carry select*, *carry skip* e de prefixo paralelo nas versões Brent-Kung, Kogge-Stone e um híbrido destas. Os somadores *carry skip* e *carry select* foram implementados utilizando cadeias de somadores de tamanhos diversos. Uma implementação de cada um dos tamanhos escolhidos (16, 32 e 64 bits) foi feita utilizando metade do comprimento em cada subcircuito de somador. E também foram utilizados somadores de 5 e 6 bits para o somador de 16 bits, 6, 7, 8, e 5 bits para o de 32 bits e 7, 8, 9, 10, 11 e 12 bits para o somador de 64 bits. A Figura 5.1 mostra a arquitetura utilizada para implementar o somador do tipo *carry look-ahead* de 64 bits.

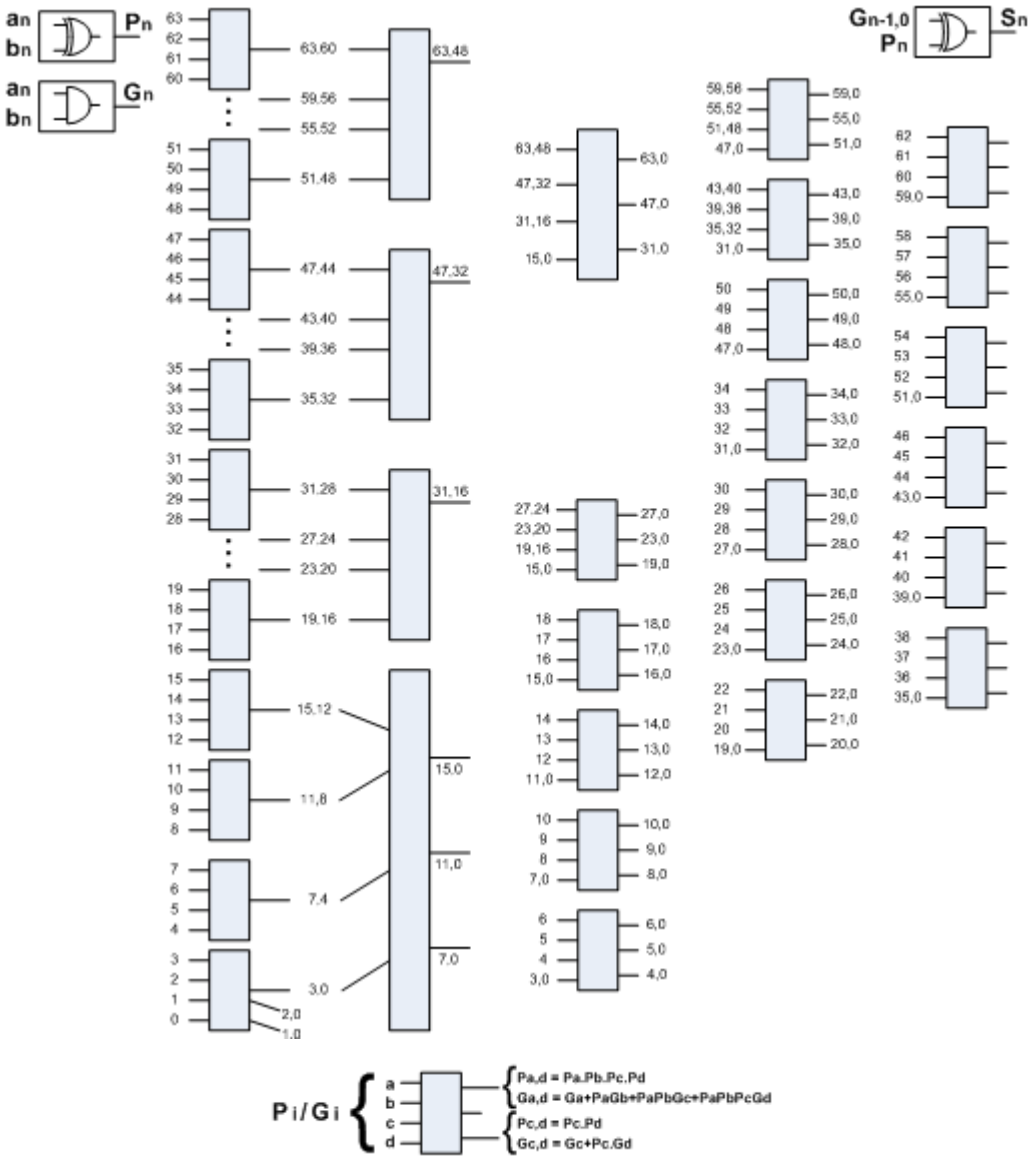

Figura 5.18: Arquitetura do somador CLA de 64 bits

5.2 Circuitos RNS

Para a implementação dos circuitos RNS, primeiramente foi escolhido o formato de base $\{2^n\text{-}1, 2^n, 2^n\text{+}1\}$. Para os circuitos de 16, 32 e 64 bits, os valores de *n* foram, respectivamente, 6, 11 e 22. Estas são bases bastante citadas em trabalhos da área para circuitos com estes tamanhos. Com isto, os operadores podem ter um ganho, o que foi comprovado por resultados preliminares deste trabalho. Um multiplicador de 64 bits, por exemplo, poderia ser implementado por três multiplicadores de 22 bits, que mesmo com os circuitos adicionais para cálculo dos módulos, pode ser vantajoso.

Entretanto, estas bases, apesar de adequadas para os valores de entrada dos operadores, não eram adequados para as saídas. Pois nas operações em RNS, temos as conversões iniciais de valores e outra após toda a computação. Durante os cálculos, todos os valores estão representados na mesma base, e consequentemente estão com a mesma abrangência, diferentemente do sistema convencional, onde um multiplicador de *n* bits tem resposta de *2n* bits, e um somador de *n* bits tem resposta de *n+1* bits.

Portanto, a escolha da base deve ser feita de acordo com a abrangência que é necessária para a saída, levando em conta não somente o tamanho das entradas, bem como o tamanho das saídas dos operadores, o que contribui para definir a representação do resultado final. Para somar dois valores de 32 bits, por exemplo, temos uma resposta de 33 bits, então a representação necessária precisa ser de 2^{33} valores, que é de 8.589.934.592 valores distintos, entretanto, com n=11, temos $(2^{11}-1) * 2^{11} * (2^{11}+1) =$ 8.589.932.544, que já não é suficiente para a abrangência desejada. Com os multiplicadores, a situação é ainda pior, pois a saída tem até o dobro de bits do que os valores da entrada. Para que os valores de saída dos operadores fossem corretos, foram escolhidos então, para os somadores de 16, 32 e 64 bits, os valores 6, 12 e 22 para *n*, e para os multiplicadores de 16, 32 e 64 bits, *n* terá os valores 11, 22 e 43. Caso a escolha da base não seja adequada para o tamanho que podemos ter no resultado final, poderá ocorrer um *overflow* deste valor, que não é detectado, e portanto não daria pra confiar no resultado final.

Os conversores foram implementados utilizando a arquitetura sugerida em (BI; JONES, 1988) e explicada no Capítulo 4 deste trabalho.

Para a implementação dos somadores, foram utilizados os somadores seguidos de um subtrator e multiplexadores. Como apenas estamos somando dois valores que são menores que o valor do modulo, o resultado nunca chegará a 2 modulo. Com isto, somente dois casos podem acontecer: ou a resposta do somador já é a resposta correta, ou este valor subtraído do valor do módulo é a resposta. Então o valor do módulo é subtraído e para definir qual valor é o correto, utiliza-se o sinal de b_{out} do subtrator. Se este for igual a um, o resultado da subtração é um valor negativo, portanto o resultado da soma era menor que o valor do módulo, então este é o valor correto. Se o b_{out} for zero, o resultado da subtração é o valor correto.

Os multiplicadores RNS foram implementados utilizando os multiplicadores seguidos de circuitos para o cálculo dos resíduos. Os resíduos são calculados por circuitos iguais aos codificadores.

5.3 Ferramentas Utilizadas

Todos os códigos foram escritos em linguagem Verilog, de forma estrutural e hierárquica. Para isto, foi utilizado o ambiente de desenvolvimento Xilinx ISE versão 8.1i, sintetizando o circuito para o dispositivo 4vlx200ff1513-11 da família Virtex 4. Para validar os circuitos, estes foram simulados utilizando o Modelsim SE 6.1 da Menthor, conforme ilustrado na Figura 5.2. Para a síntese para ASIC, foram utilizadas as ferramentas da Synopsys, como o Design Compiler. Um exemplo de leiaute do ASIC gerado pelas ferramentas da Synopsys é mostrado na Figura 5.3.

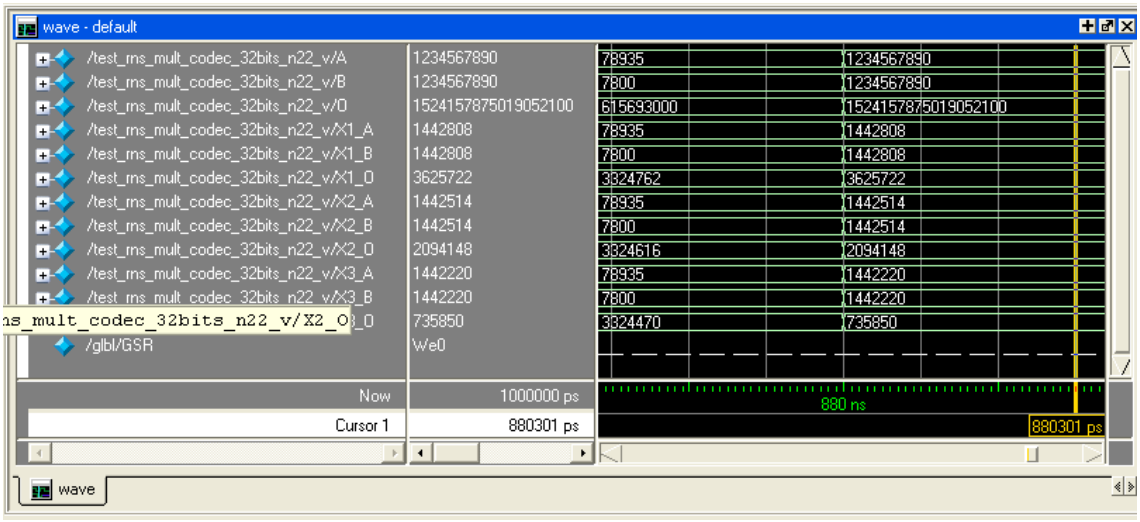

Figura 5.19: Simulação no ModelSim.

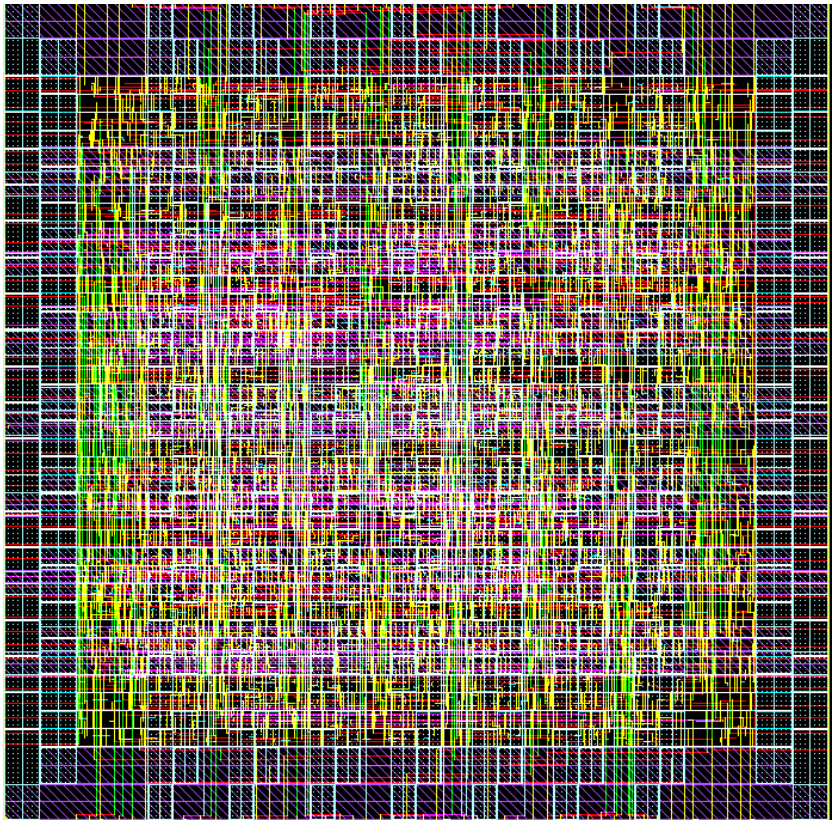

Figura 5.20: Leiaute do somador CLA gerado pela ferramenta da Synopsys.

5.4 Resultados e Análise

5.4.1 Somadores 16 bits

Na Tabela 5.1 podemos ver o resultado para a síntese da Synopsys para o somador de 16 bits. Para este operador, a implementação que gerou o menor número de células. ocupou a menor área e teve menor consumo de potência foi o somador *ripple carry*. A menor estimativa de atraso foi do somador paralelo de prefixo Kogge-Stone. Estes

resultados confirmam as espectativas sobre os somadores, visto que o somador do tipo *ripple carry* é o mais simples, mas com maior cadeia de *carry*, enquanto que o somador Kogge-Stone tem uma estrutura que otimiza bastante o tempo de atraso.

Tabela 5.6: Resultados Synopsys para síntese do somador de 16 bits

Circuito	Número de células	Área	Atraso (ns)	Potência (W)	*Leakage* (nW)
Ripple	47	733,65	2,91	102,18	204,11
Look-Ahead	89	1180,64	1,30	152,75	325,38
Select (8-8)	111	1527,66	2,37	231,90	453,70
Select (5-5-6)	131	1772,84	1,83	271,98	529,13
Skip (8-8)	74	980,72	4,38	138,20	280,59
Skip (5-5-6)	80	1014,67	4,41	147,70	293,06
PPA Brent-Kung	99	1286,25	1,12	167,69	352,32
PPA Kogge-Stone	145	1850,17	0,84	245,80	491,31
PPA híbrido	111	1433,36	0,97	185,39	387,02
+	66	875,10	4,12	128,13	254,26
RNS (n=6, sem conversores)	179	2101,00	2,34	372,26	628,97
RNS (n=6, com conversores)	828	9535,62	12,24	2078,3	2840,9

Na Tabela 5.2 são mostrados os resultados da Xilinx para o somador de 16 bits. A implementação utilizando o operador '+' foi a que apresentou os melhores resultados de área e atraso.

Tabela 5.7: Resultados Xilinx do somador de 16 bits

Circuito	Número de Slices	Número de LUTs	Atraso (ns)
Ripple	18	31	16,947
Look-Ahead	26	46	14,495
Select (8-8)	17	33	7,504
Select (5-5-6)	22	40	8,251
Skip (8-8)	11	22	8,341
Skip (5-5-6)	14	24	9,933
PPA Brent-Kung	21	37	15,613
PPA Kogge-Stone	32	55	13,310
PPA híbrido	29	50	14,260
+	8	16	6,918
RNS (n=6, sem conversores)	22	39	9,581
RNS (n=6, com conversores)	125	223	26,254

5.4.2 Somadores de 32 bits

A Tabela 5.3 mostra os resultados da Synopsys para a síntese do somador de 32 bits. A implementação que apresentou o menor número de células, de área e de consumo de potência foi novamente a do *ripple carry*. A menor estimativa de atraso foi novamente do paralelo de prefixo Kogge-Stone.

Tabela 5.8: Resultados Synopsys para síntese do somador de 32 bits

Circuito	Número de células	Área	Atraso (ns)	Potência (W)	*Leakage* (nW)
Ripple	95	1488,05	5,92	210,14	412,31
Look-Ahead	192	2529,13	1,46	327,98	694,46
Select (16-16)	215	3036,46	4,70	466,33	894,66
Select (6-6-7-8-5)	280	3847,44	2,65	598,47	1143,6
Skip (16-16)	144	1961,44	8,83	273,70	553,50
Skip (6-6-7-8-5)	162	2084,03	8,90	302,54	589,20
PPA Brent-Kung	209	2710,18	1,45	350,80	736,12
PPA Kogge-Stone	353	4475,48	1,00	572,65	1157,0
PPA híbrido	255	3274,10	1,12	412,98	865,71
+	130	1780,38	8,57	256,16	514,79
RNS (n=12, sem conversores)	331	4100,16	4,17	678,13	1214,5
RNS (n=12, com conversores)	1570	18739,30	22,84	3957,5	5540,5

A Tabela 5.4 mostra os resultados da Xilinx para o somador de 32 bits. O circuito implementado para o operador '+' foi novamente o mais vantajoso em área e atraso.

Tabela 5.9: Resultados Xilinx do somador de 32 bits

Circuito	Número de Slices	Número de LUTs	Atraso (ns)
Ripple	36	63	29,715
Look-Ahead	59	102	18,932
Select (16-16)	34	65	7,920
Select (6-6-7-8-5)	48	88	10,136
Skip (16-16)	23	44	9,427
Skip (6-6-7-8-5)	28	51	13,070
PPA Brent-Kung	54	94	23,025
PPA Kogge-Stone	109	190	15,575
PPA híbrido	70	122	14,667
+	16	32	7,526
RNS (n=12, sem conversores)	48	78	9,306
RNS (n=12, com conversores)	201	305	26,011

5.4.3 Somadores de 64 bits

A Tabela 5.5 mostra os resultados da síntese da Synopsys para o somador de 64 bits. O circuito *ripple carry* foi o que obteve menor número de células, menor área e menor consumo de potência. O paralelo de prefixo Kogge-Stone foi o que obteve menor estimativa de atraso. Estes valores confirmam as comparações já feitas para os outros tamanhos de somadores.

Tabela 5.10: Resultados Synopsys para síntese do somador de 64 bits

Circuito	Número De células	Área	Atraso	Potência (W)	*Leakage* (W)
Ripple	191	2996,85	11,92	425,93	0,83
Look-Ahead	389	5169,53	2,12	669,73	1,41
Select (32-32)	423	6054,06	9,34	941,39	1,78
Select (7-7-8-9-10-	573	8034,36	3,66	1262,6	2,38

11-12)					
Skip (32-32)	282	3904,02	17,72	544,11	1,10
Skip (7-7-8-9-10-11-12)	312	4130,34	17,84	598,24	1,16
PPA Brent-Kung	431	5582,56	1,80	718,48	1,51
PPA Kogge-Stone	833	10510,68	1,16	1293,4	2,66
PPA híbrido	575	7347,86	1,29	903,47	1,91
+	258	3590,94	17,46	511,46	1,04
RNS (n=22, sem conversores)	591	7457,24	7,20	1192,3	2,19
RNS (n=22, com conversores)	2850	34242,21	42,55	7594,9	10,03

A Tabela 5.6 mostra os resultados da Xilinx para o somador de 64 bits. O circuito implementado para o operador '+' foi novamente o mais vantajoso em área e atraso.

Tabela 5.11: Resultados Xilinx do somador de 64 bits

Circuito	Número de Slices	Número de LUTs	Atraso (ns)
Ripple	73	127	55,251
Look-Ahead	125	218	24,694
Select (32-32)	67	129	8,868
Select (7-7-8-9-10-11-12)	100	184	12,331
Skip (32-32)	45	87	10,193
Skip (7-7-8-9-10-11-12)	55	101	17,068
PPA Brent-Kung	123	214	31,275
PPA Kogge-Stone	281	489	17,569
PPA híbrido	175	305	16,753
+	32	64	8,742
RNS (n=22, sem conversores)	85	138	9,893
RNS (n=22, com conversores)	360	546	28,921

5.4.4 Multiplicadores de 16 bits

A Tabela 5.7 mostra os resultados de síntese da Synopsys para o multiplicador de 16 bits. O menor número de células e área foram do multiplicador do tipo array. O menor atraso foi do circuito implementado pelo operador *. O menor consumo de potência foi do multiplicador modular aditivo.

Tabela 5.12: Resultados Synopsys para síntese do multiplicador de 16 bits

Circuito	Número de células	Área	Atraso (ns)	Potência (mW)	*Leakage* (W)
Array	976	14212,90	7,63	2,18	4,25
Modular	1580	17811,38	8,65	2,76	5,14
Modular aditivo	1024	14967,30	11,62	2,16	4,50
*	1557	16031,00	5,74	3,17	4,69
RNS (n=11, sem conversores)	2503	26701,99	12,37	5,18	7,74
RNS (n=11, com conversores)	3463	37650,22	27,71	8,04	10,96

A Tabela 5.8 mostra os resultados da Xilinx para o multiplicador de 16 bits. A área não pode ser facilmente comparada, pois componentes diferentes foram utilizados. O operador * foi o que apresentou o menor atraso.

Tabela 5.13: Resultados Xilinx do multiplicador de 16 bits

Circuito	Número de Slices	Número de LUTs	Número de DSP48s	Atraso (ns)
Array	285	495	0	29,988
Modular	236	426	0	15,983
Modular aditivo	314	548	0	40,262
*	0	0	1	9,309
RNS (n=11, sem conversores)	42	58	3	15,894
RNS (n=11, com conversores)	149	211	3	29,346

5.4.5 Multiplicadores de 32 bits

A Tabela 5.9 mostra os resultados de síntese da Synopsys para o multiplicador de 32 bits. O multiplicador do tipo *array* foi o que apresentou menor número de células e menor atraso. O modular aditivo apresentou menor consumo de potência. O operador '*' foi o que apresentou menor atraso.

Tabela 5.14: Resultados Synopsys para síntese do multiplicador de 32 bits

Circuito	Número de células	Área	Atraso	Potência (mW)	*Leakage* (W)
Array	4000	58360,39	15,63	9,56	17,36
Modular	6604	75357,02	17,89	11,93	21,79
Modular aditivo	4096	59869,19	31,08	9,17	17,87
*	6709	67107,66	10,98	14,25	20,04
RNS (n=22, sem conversores)	10058	102860,55	25,32	21,52	30,50
RNS (n=22, com conversores)	11957	124666,48	55,35	30,52	36,93

A Tabela 5.10 mostra os resultados da Xilinx para o multiplicador de 32 bits. A área não pode ser facilmente comparada, pois componentes diferentes foram utilizados. O operador * foi o que apresentou o menor atraso.

Tabela 5.15: Resultados Xilinx do multiplicador de 32 bits

Circuito	Número de Slices	Número de LUTs	Número de DSP48s	Atraso (ns)
Array	1159	2015	0	56,365
Modular	986	1782	0	19,415
Modular aditivo	1311	2280	0	97,832
*	0	0	4	15,544
RNS (n=22, sem conversores)	80	113	11	23,179
RNS (n=22, com conversores)	287	408	11	38,939

5.4.6 Multiplicadores de 64 bits

A Tabela 5.11 mostra os resultados da síntese da Synopsys para o multiplicador de 64 bits. Aqui as tendências apresentadas nos multiplicadores anteriores se confirmam.

Tabela 5.16: Resultados Synopsys para síntese do multiplicador de 64 bits

Circuito	Número de células	Área	Atraso	Potência (mW)	*Leakage* (W)
Array	16192	236459,14	31,61	40,05	70,15
Modular	26988	309764,19	36,02	49,57	89,66
Modular aditivo	16384	239476,75	79,88	37,84	71,15
*	27498	273439,84	21,59	59,64	81,94
RNS (n=43, sem conversores)	38650	386143,41	47,76	84,07	115,12
RNS (n=43, com conversores)	42357	428717,97	100,47	113,29	127,67

A Tabela 5.18 mostra os resultados da Xilinx para o multiplicador de 64 bits. A área não pode ser facilmente comparada, pois componentes diferentes foram utilizados. O operador * foi o que apresentou o menor atraso.

Tabela 5.17: Resultados Xilinx do multiplicador de 64 bits

Circuito	Número de Slices	Número de LUTs	Número de DSP48s	Atraso (ns)
Array	4673	8127	0	109,119
Modular	4027	7286	0	24,162
Modular aditivo	5345	9296	0	238,508
*	79	157	16	20,602
RNS (n=43, sem conversores)	261	427	24	28,507
RNS (n=43, com conversores)	656	996	24	47,102

5.4.7 Análise Comparativa

Como podemos ver nos resultados acima, os operadores em RNS não apresentam vantagens sobre aqueles em notação convencional, mesmo sem considerarmos os conversores. Se os circuitos RNS sem os conversores tivessem se mostrado vantajosos, mesmo que o operador com conversores não apresentasse vantagem, ainda assim o RNS poderia ser vantajoso, pois se poderia utilizar vários operadores RNS em seqüência, sem conversores entre estes operadores, somente nas entradas primárias e nos resultados. A necessidade de se usar sempre a mesma base para todos os valores a serem representados é um grande obstáculo para a utilização do RNS, pois os operadores acabam se tornando circuitos menos vantajosos, visto que mesmo que as entradas estejam com a base adequada, a saída estará representada na mesma base, e se a base não for adequada ao tamanho da saída, poderá ocorrer um erro no resultado devido a um *overflow* ocorrido e não detectado.

Nos multiplicadores sintetizados na ferramenta da Xilinx, a comparação dos resultados ficou um pouco prejudicada, visto que os circuitos que eram descritos utilizando o operador "*" e os circuitos RNS utilizam, além das LUTs, elementos DSP48, que as outras descrições não utilizam. Mesmo assim, é possível verificar que os

circuitos RNS, mesmo sem os conversores, utilizam mais elementos DSP48 que o circuito do operador, e mesmo assim, utiliza mais LUTs e tem maior atraso.

Nos resultados da Synopsys para circuitos implementados somente utilizando-se o operador *, estes apresentaram, em alguns casos, uma vantagem muito além da esperada sobre os demais circuitos. A hipótese mais provável para justificar esta ocorrência é que o comportamento padrão da ferramenta é substituir blocos que ela reconheça por IPs pré-definidos e otimizados.

CONCLUSÃO

Neste trabalho, foram abordados diversos sistemas de representação numérica e suas características principais. O sistema numérico por resíduos acabou sendo escolhido para ser estudado com mais profundidade, devido ao grande número de publicações atuais sobre o assunto. Além disso, foram abordadas também as principais arquiteturas de somadores e multiplicadores do sistema binário.

Trabalhos de diversos autores apontavam para bons resultados na implementação de circuitos aritméticos utilizando o sistema numérico por resíduos. Os resultados obtidos neste trabalho, entretanto, não confirmaram esta vantagem. Ainda seria necessário rever as implementações dos circuitos e tentar otimizar as estruturas de cálculo em RNS, para analisar se eles realmente podem apresentar vantagens em relação aos circuitos tradicionais.

O custo causado pelas conversões de binário para RNS e de RNS para binário foi consideravelmente grande, provocando um impacto negativo nas características do circuito final. Além disso, a necessidade de escolha de valores altos para a base para garantir que não houvesse nenhum *overflow* na saída das operações também contribuiu para que os subcircuitos para o cálculo de cada resíduo não apresentasse muita vantagem em relação à versão para o sistema binário.

Finalmente, pode-se concluir que o sistema numérico por resíduos possui alguma vantagem para a implementação de circuitos aritméticos, como o maior paralelismo. Mas seu uso precisa ser muito bem planejado, pois os resultados obtidos neste trabalho não demonstram vantagens significativas nas implementações analisadas.

REFERÊNCIAS

BI, G.; JONES, E. V. Fast conversion between binary and residue numbers. **Electronics Letters**, [S.l.], v.24, n.19, p.1195-1197, 1988.

KOREN, I. **Computer Arithmetic Algorithms** 2nd ed. Natick, Massachusetts: A K Peters, 2002.

LIU, J. et al. An algorithmic approach for generic parallel adders. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2003. **Proceedings**. . . [S.l.: s.n.], 2003. p.734-740.

LU, M. **Arithmetic and Logic in Computer Systems**. New Jersey: John Wiley & Sons, 2004.

MOHAN, P. V. A. **Residue Number Systems**: algorithms and architectures. [S.l.]: Kluwer Academic Publishers, 2002.

PARHAMI, B. **Computer Arithmetic**: algorithms and hardware designs. New York: Oxford University Press, 2000.

PREMKUMAR, A. B. A formal Framework for Conversion From Binary to Residue Numbers. **IEEE Transactions on Circuits and Systems - II: Analog and Digital Signal Processing**, [S.l.], v.49, n.2, p.135-144, Feb. 2002.

RE, M.; et al. FPGA Realization of RNS to Binary Signed Digit Architecture. In: **IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS**. ISCAS, 2001, Sydney, NSW. **Proceedings**. . . [S.l.: s.n.], 2001. v.4, p.350-353.

SOUDRIS, D. J.; et al. A CAD Tool for Architecture Level Exploration and Automatic Generation of RNS Converters. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2001. **Proceedings**. . . [S.l.: s.n.], 2001. v.4, p.730-733.

WANG, W.; SWAMY, M. N. S.; AHMAD, M. O. Moduli Selection in RNS for Efficient VLSI Implementation. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2003. **Proceedings**. . . [S.l.: s.n.], 2003. v.4, p.512_515.

WANG, W.; et al. A Study of the Residue-to-Binary Converters for the Three-Moduli Sets. **IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications**, [S.l.], v.50, n.2, p.235-243, Feb. 2003.

WEBER, R. F. **Fundamentos de Arquitetura de Computadores**. Porto Alegre, RS: Sagra Luzzatto, 2004.

WEI, S. Number Conversions between RNS and Mixed-Radix Number System Based on Modulo (2^p - 1) Signed-Digit Arithmetic. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, **Proceedings**… Florianópolis, Brazil, p.160-165.