Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Programa de Pós-Graduação em Engenharia Elétrica

Dissertação de Mestrado

Nova Arquitetura de Multiplicador em GF (2^8) Utilizando Portas de Limiar Linear

Mestrando

Marlo Andrade Santos

Orientador

Prof. Dr. Raimundo Carlos Silvério Freire

Orientador

Prof. Dr. Francisco Marcos de Assis

Campina Grande – Paraíba – Brasil,

Setembro de 2015

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Engenharia Elétrica

Nova Arquitetura de Multiplicador em GF (2^8) Utilizando Portas de Limiar Linear

Dissertação apresentada à Coordenadoria do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, em cumprimento às exigências para obtenção do Grau de Mestre no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Marlo Andrade Santos

Orientador

Prof. Dr. Raimundo Carlos Silvério Freire

Orientador

Prof. Dr. Francisco Marcos de Assis

Campina Grande – Paraíba – Brasil,

Setembro de 2015

Dedicatória

Dedico este trabalho aos meus pais, Antônio Bento e Maria Gilza Andrade, e aos meus irmãos, Gledson Andrade e Shirley Andrade.

Agradecimentos

Agradeço aos meus orientadores Professores Raimundo Carlos Silvério Freire e Francisco Marcos de Assis, pela oportunidade de trabalhar e desenvolver atividades de pesquisa científica no Laboratório de Instrumentação e Metrologia Científicas, LIMC, e no Instituto de Estudos em Computação e Informação Quânticas, IQuanta, na Universidade Federal de Campina Grande.

A Vanderson Reis, Arthur Farias, Nelson Campos e Marconni Menezes.

Aos meus colegas da pós-graduação do Laboratório de Instrumentação e Metrologia Científicas.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico e do Programa de Pós-Graduação em Engenharia Elétrica PPgEE-Copele.

Resumo

Nesta dissertação são apresentados o desenvolvimento e implementação em *hardware* de uma nova arquitetura de multiplicador em corpos finitos baseada no multiplicador de Mastrovito. Nesta arquitetura são utilizadas as portas de limiar linear como elemento básico de processamento, que é o elemento básico de uma rede neural discreta. As redes neurais discretas implementadas com portas de limiar permitem reduzir a complexidade dos circuitos quando comparados com implementações com lógica tradicional (portas AND, OR e NOT). Por esta razão, estender e implementar portas de limiar linear na aritmética dos corpos finitos se torna atraente. Assim, com a finalidade de comprovar a eficiência de tais portas como unidades básicas de processamento da arquitetura de multiplicadores em $GF(2^n)$, foi projetado, na linguagem de descrição de *hardware* Verilog, um multiplicador em $GF(2^8)$ utilizando portas de limiar linear. Foram desenvolvidos diversos níveis de abstração e utilizado a FPGA (*Field-Programmable Gate Array*), ferramenta Quartus II® e a placa de desenvolvimento EP2C35F672C6, da Altera®. Os resultados do desenvolvimento são apresentados. A partir deles é apresentado o funcionamento prático da nova arquitetura proposta do multiplicador em $GF(2^8)$. A partir dos resultados da operação de multiplicação em corpos finitos, observou-se uma taxa de acerto de 90%., verificando-se, entretanto, que o tempo de processamento e contagem de portas ficou abaixo do valor esperado.

Palavras-chave: corpo finito, multiplicador, portas de limiar linear.

Abstract

This dissertation describes the design, the developing and the implementation in hardware of a new architecture of multiplying finite fields based upon the Mastrovito multiplier. Such architecture utilizes linear threshold ports as basic processing elements, which are the basic elements of a discrete neural network. The discrete neural networks implemented with threshold ports allow reduce the complexity of the circuits when they are compared to implementations of traditional logics (AND, OR and NOT ports). For this reason, extending and implementing linear threshold ports in the arithmetic's of the finite fields becomes an attractive activity. Thus, with the objective of proving the efficiency of such ports as basic units of processing of the multiplying architecture in $GF(2^n)$, that it has been designed, in the hardware description language Verilog, a $GF(2^8)$ multiplier utilizing the linear threshold ports. Several levees of abstraction have been developed. The FPGA (Field-Programmable Gate Array) Quartus II® tool and the developing Altera® hardware EP2C35F672C6 have been utilized. The results of the development which are presented indicate the practical functioning of the new architecture proposed by the $GF(2^8)$ multiplier. However, its efficiency in terms of time processing and counting of ports is under what would be expected. From the results the multiplication operation in finite fields was observed with an accuracy rate of 90%.

Key words: finite fields, multiplier, linear threshold gates.

Índice

Índice de Figuras

Índice de Abreviações

GF – *Galois Field*

FPGA - *Field-Programmable Gate Array*

ASIC - *Application-Specific Integrated Circuit*

CLB - *Configuration Logical Blocks*

RAM - (*Random Access Memory*

DSP - *Digital Signal Processor*

PLL - *Phase-Locked Loop*

DLL - *Delay-Locked Loop*

PCI - *Peripheral Component Interconnect*

AON – AND, OR and NOT

VLSI - *Very Large Scale Integration*

MOS - *Metal Oxide Semiconductor*

CMOS - *Complementary Metal Oxide Semiconductor*

NIST - *National Institute of Standards and Technology*

AON-T: Arquitetura Tradicional

AON-TG: Arquitetura Tradicional com Portas de Limiar

MAS-TG: Arquitetura de Mastrovito com Portas de Limiar

ARQ-PRO: Nova Arquitetura proposta com Portas de Limiar Linear

Índice de Quadros

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S237n Santos, Marlo Andrade.

Nova arquitetura de multiplicador em GF (2^8) utilizando portas de limiar linear / Marlo Andrade Santos. -- Campina Grande, 2015.

134 f. : il.

Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2015.

"Orientação: Prof. Dr. Raimundo Carlos Silvério Freire, Prof. Dr. Francisco Marcos de Assis".

Referências.

1. Multiplicador - Engenharia Elétrica. 2. Corpo Finito. 3. Portas de Limiar Linear. I. Freire, Raimundo Carlos Silvério. II. Assis, Francisco Marcos de. III. Título.

CDU 621:004.315.5(043)

1. INTRODUÇÃO

1. Introdução

1.1 Motivação

Os corpos finitos ou campos de *Galois*, tem se mostrado útil no desenvolvimento de soluções em diversas áreas da engenharia, como na codificação para controle de erros, em criptografia, em processamento digital de sinais, entre outras. Operações aritméticas em corpos finitos, tais como soma, multiplicação, divisão, cálculo de inversa e exponenciação, são de fundamental importância para as áreas mencionadas. Por exemplo, as operações de multiplicação e divisão são utilizadas nos codificadores e decodificadores em sistemas RS de correção de erros. Por sua vez, a soma, a multiplicação, o cálculo de inversa e exponenciação são fundamentais para o desenvolvimento de criptosistemas que operam com curvas elípticas [1].

Assim, são desejáveis multiplicadores em corpos finitos cada vez mais eficientes para a melhoria do desempenho dos sistemas que os utilizam em virtude da demanda por altas velocidades para esses sistemas [2].

Multiplicadores podem ser desenvolvidos em *software* e *hardware*. Implementações em *software* tem a desvantagem da necessidade de grande quantidade de memória e capacidade de armazenamento [3]. Pesquisas tem sido realizadas em busca de métodos de multiplicação em corpos finitos, implementados em *hardware* mais eficientes e velozes. Esta eficiência depende fundamentalmente da forma de representação do corpo [4].

Projetos em FPGA(*Field Programmable Gate Array*) têm sido amplamente realizados [5-6]. Muitos deles são projetados utilizando portas AND, OR e NOT, ditas portas tradicionais. Uma alternativa para obtenção de arquiteturas de multiplicadores mais eficientes é a utilização de portas com um potencial maior que as portas AON.

Os resultados teóricos obtidos em [7] demonstraram que a utilização de portas de limiar linear em arquiteturas de multiplicação em corpos finitos é justificada, já que diminui consideravelmente a complexidade espacial e temporal do circuito do multiplicador se comparado ao multiplicador com portas tradicionais. Desta forma, torna-se necessário à implementação física da arquitetura proposta para comprovar sua viabilidade.

A multiplicação é uma operação elementar em diversas áreas da engenharia, destacando-se em processamento digital de sinais, criptografia e códigos corretores de erros. Além das aplicações descritas, é uma operação utilizada para implementar outras operações mais complexas, como a exponenciação e divisão.

O uso da matemática, em especial da álgebra de corpos finitos (ou campo de *Galois*) desenvolvida pelo francês *Évariste Galois*, vem sendo de fundamental importância para o desenvolvimento de várias áreas da engenharia, em especial a criptografia e os códigos corretores de erros [8].

Modalidades importantes de processamento de sinais tais como codificação para proteção de erros e a criptografia exigem a realização de operações aritméticas envolvendo elementos de corpos finitos. Em alguns casos, dependendo da velocidade de processamento que é necessário, utiliza-se *hardware* dedicado ou uma solução mista usando *DSP's* combinados com *hardware* dedicado [9].

Arquiteturas que fazem o uso das redes neurais tem se destacado em diversas aplicações [1-7-4-5-6-8], no que diz respeito à complexidade computacional dos circuitos. O elemento de processamento destas redes são portas que calculam uma função de limiar linear ou um elemento analógico que executa uma função *sigmoidal* [7]. Segundo [10-11], estas portas demonstram um excelente potencial, quando comparadas com as portas tradicionais, em relação à redução de área do *chip* e obtenção de maiores velocidades de operação.

1.2 Objetivo da Dissertação

O objetivo desta dissertação de mestrado consiste no estudo, desenvolvimento e implementação de um circuito que realize a multiplicação em corpos finitos utilizando portas de limiar linear como elemento básico de processamento. O multiplicador proposto e implementado em FPGA é baseado na arquitetura de Mastrovito, pois esta apresenta uma das melhores medidas de complexidade espacial quando comparada com outras arquiteturas de multiplicadores paralelos de alto desempenho.

1.3 Organização do Trabalho

Esta dissertação está organizada da seguinte maneira: no capítulo 1 é realizada uma breve introdução, onde é apresentada a motivação, objetivos e organização do trabalho.

No capítulo 2 é abordada a aritmética em corpos finitos, bem como uma revisão sobre estruturas algébricas e suas propriedades e a apresentação do multiplicador de Mastrovito.

No capítulo 3 são apresentadas as redes neurais discretas e algumas características e propriedades das portas de limiar linear, dos circuitos de limiar linear e da construção de circuitos de 2 e 3 camadas.

No capítulo 4 é apresentada uma nova arquitetura para multiplicadores em $GF(2^4)$ e $GF(2^8)$ utilizando portas de limiar linear.

No capítulo 5 são expostas e discutidas diferentes implementações destes multiplicadores, baseadas no multiplicador de Mastrovito e utilizando portas de limiar linear.

Por fim, no capítulo 6 são apresentadas as considerações finais e sugestões de trabalhos futuros para continuação da pesquisa.

2. ESTRUTURAS ALGÉBRICAS

2. Aritmética de $GF(2^n)$ em *Hardware*

2.1. Álgebra de Corpos Finitos

Unidades aritméticas em corpos finitos, como multiplicadores, somadores e subtratores em $GF(p^n)$, se baseiam em estruturas algébricas e na teoria dos corpos finitos. O objetivo deste capítulo é fornecer os fundamentos básicos dessa álgebra, com a finalidade de facilitar o entendimento do trabalho. Uma descrição pormenorizada sobre a teoria dos corpos finitos é encontrada nas referências [12-16].

2.1.1 Propriedades Básicas

A álgebra possui três estruturas básicas chamadas de: corpo, anel e grupo. A partir desta construção, toda a teoria da Aritmética de Corpos Finitos é desenvolvida. As estruturas consistem em conjuntos de objetos matemáticos, como os números inteiros e números reais, que são associados a regras de relação entre os elementos que os constituem.

Definição 2.1 (Grupo) Um conjunto G associado a uma operação binária $G + G \rightarrow G$ é denominado um grupo se as condições seguintes forem satisfeitas:

- A operação binária + é associativa, ou seja, $(a + b) + c = a + (b + c)$, para todo $a, b, c \epsilon$ G;
- Existe um único elemento identidade e este pertence a G, tal que $a + e = e + a = a$, para todo $a \epsilon$ G;
- Existe um único elemento inverso $a' \epsilon$ G, tal que $a + a' = a' + a = e$, para todo elemento $a \epsilon$ G.

Se um grupo satisfaz a condição de que $a + b = b + a$, para todo $a, b \epsilon$ G, diz-se que o grupo é *abeliano* ou *comutativo.*

Definição 2.2 (Anel) Um conjunto R associado a duas operações binárias $R + R \rightarrow R \, e \, R \times R \rightarrow R$ é denominado anel se as condições seguintes forem satisfeitas:

- R constitui um grupo abeliano sob a operação binária +, chamada adição. O elemento identidade com relação à adição é o 0, chamado elemento zero;

- A operação binária ×, chamada multiplicação, é associativa, ou seja, $a \times (\mathrm{b} \times \mathrm{c}) = (a \times \mathrm{b}) \times \mathrm{c}$, para todo $a, b, c \in \mathrm{R}$;
- A operação binária × obedece a propriedade da distribuição, ou seja, $a \times (\mathrm{b} + \mathrm{c}) = a \times \mathrm{b} + a \times \mathrm{c}$ e $(\mathrm{b} + \mathrm{c}) \times a = b \times a + \mathrm{c} \times a$, para todo $a, b, c \in \mathrm{R}$.

Um anel é *comutativo* quando a multiplicação é comutativa, quando $a \times \mathrm{b} = \mathrm{b} \times a$, para todo $a, b \in \mathrm{R}$. Um anel não necessariamente possui identidade na multiplicação, assim como inversas de seus elementos. Caso um anel tenha identidade na multiplicação, esta identidade é única. Além disso, se $a \times \mathrm{b} = 1$ e $\mathrm{c} \times a = 1$, então $b = c$ e a é dito ter uma única inversa denotada por a^{-1}. Um elemento que possua inversa em um anel é dito uma *unidade* [17].

Definição 2.3 (Corpo) Um conjunto F associado a duas operações binárias $F + F \rightarrow F$ e $F \times F \rightarrow F$ é denominado corpo se as condições seguintes forem satisfeitas:

- F constitui um grupo abeliano sob a operação binária +, chamada adição. O elemento identidade com relação à adição é o 0, chamado elemento zero;
- O conjunto dos elementos de F, excluindo o zero, forma um grupo sob a operação binária ×, chamada multiplicação. O elemento identidade com relação a multiplicação é o 1, chamado elemento unidade;
- A operação de multiplicação é distributiva sobre a adição. Ou seja, para α, β e $\gamma \in$ F, tem-se que $\alpha \times (\beta + \gamma) = \alpha \times \beta + \alpha \times \gamma$.

Por exemplo, o conjunto $\{0,1,2,3,4,5,6\}$ com adição e multiplicação módulo 7 forma um corpo, pois:

1) O conjunto é fechado sobre a adição módulo 7 .

$$2 + 5 = 7 \mod 7 = 0 \tag{2.1}$$

$$3 + 5 = 8 \mod 7 = 1 \tag{2.2}$$

2) O conjunto de todos os elementos diferentes de zero é fechado sobre a multiplicação módulo 7.

$$34 = 12 \mod 7 = 5 \tag{2.3}$$

$$52 = 10 \mod 7 = 3 \tag{2.4}$$

3) Satisfaz a propriedade da distribuição, pois:

$$3(4+4) = 27 \mod 7 = 6 \tag{2.5}$$

$$34 + 35 = 12 + 15 = 27 \mod 7 = 6 \tag{2.6}$$

Também são exemplos de corpos os conjuntos dos números reais, dos números complexos e dos números racionais. Os corpos com número finito de elementos são chamados de *corpos finitos* ou *corpos de Galois*. Um corpo finito com q elementos é denotado por $GF(q)$. Um subconjunto de um corpo é dito subcorpo se constituir um corpo sob as mesmas operações inerentes ao conjunto. Assim, o corpo é dito uma extensão deste subcorpo.

A ordem q de um corpo é uma potência de um número primo, ou seja, $q = p^n$, sendo p primo. Além disso, existe sempre um único corpo de ordem p^n para qualquer primo p e n inteiro e positivo.

Definição 2.4 (Corpos Finitos) Corpos Finitos são denotados por $GF(q)$, onde q é um número primo positivo e o conjunto $\{0, 1, 2, \ldots, q-1\}$ e as operações binárias $\times$ e + são a multiplicação e adição módulo q, respectivamente.

Em Sistemas Digitais são utilizados bits sequenciados a partir de 0 e 1, então o interesse é maior na subclasse de corpos finitos de q tal que este seja potência de 2. O menor inteiro positivo λ para o qual $\sum_{i=1}^{\lambda} 1 = 0$ em um corpo é dito ser a *característica do corpo.*

Circuitos Digitais operam naturalmente sobre base binária e as arquiteturas de implementação de *hardware* são baseadas em corpos de característica 2. Ou seja, para estes sistemas o foco é em corpos de característica 2. Em $GF(2)$, a representação do corpo finito é mapeada convenientemente no domínio digital [1]. Utilizando a notação $GF(q)$, a subclasse será denotada por $GF(2^n)$. Portanto, $GF(2)$, $GF(4)$, $GF(8)$, $GF(16)$, etc são corpos finitos $GF(2^n)$.

O campo de Galois $GF(q)$, em que p é primo, é chamado de corpos primos. Qualquer corpo no qual p seja primo pode ter as operações de adição e multiplicação

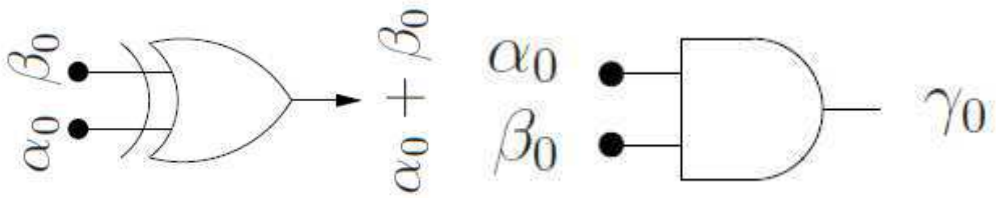
módulo p como duas operações binárias. O campo de Galois $GF(p^n)$ é chamado de extensão de corpo com $GF(q)$, sendo corpo de base de $GF(p^n)$. O campo de Galois $GF(2)$ é chamado de corpo binário, pois contém o conjunto $\{0,1\}$ e utiliza adição e multiplicação módulo 2. No Quadro 2.1 temos as operações básicas de adição e multiplicação em $GF(2)$ e na Figura 2.1 os símbolos das portas equivalentes das duas operações.

Quadro 2.1 - Operações básicas de adição e multiplicação em GF(2)

Multiplicação *mod* 2	Adição *mod* 2
$0 \times 0 = 0$	$0 + 0 = 0$
$0 \times 1 = 0$	$0 + 1 = 1$
$1 \times 0 = 0$	$1 + 0 = 1$
$1 \times 1 = 1$	$1 + 1 = 0$

A multiplicação *mod* 2 é equivalente a porta AND e a adição *mod* 2 é equivalente a porta XOR.

Figura 2.1 - Portas lógicas para operações de adição e multiplicação em $GF(2)$



O polinômio em $GF(2)$ pode ser representado como coeficientes de $GF(2)$ $a_0 + a_1x^1 + a_2x^2 + a_3x^3 + \cdots + a_nx^n$, em que $a_i = 0\ ou\ 1$ para $0 \leq i \geq n$. Um polinômio é de grau n se não existir nenhum termo x no polinômio com potência maior que n.

2.1.2 Propriedades dos Corpos Finitos

Corpos finitos podem ser obtidos a partir de anéis polinomiais. Considere $F[x]$, um anel polinomial sobre o corpo F. Fazendo a escolha de qualquer polinômio $p(x)$ que pertença a $F[x]$, temos:

Teorema 2.1 *Para qualquer polinômio mônico em F, o anel dos polinômios módulo* $p(x)$ *é o conjunto de todos os polinômios com grau menor que* $p(x)$ *junto com a adição e multiplicação polinomial módulo* $p(x)$. *Se o polinômio* $p(x)$ *for primo, o anel de polinômios é um corpo* [1].

Assim, encontrando um polinômio primo $p(x)$ de grau n em $GF(p)$, pode-se construir um corpo de Galois com p^n elementos. Um polinômio $p(x)$ de grau n sobre $GF(2^n)$ é um polinômio na forma:

$$p(x) = p_0 + p_1x^1 + p_2x^2 + p_3x^3 \ldots + p_nx^n, \tag{2.7}$$

em que os coeficientes p_i , para $0 \leq i \geq n$, são elementos de $GF(2) = \{0,1\}$.

Polinômios em $GF(2)$ podem ser adicionados, multiplicados, subtraídos e divididos na forma padrão.

Teorema 2.2 *Um polinômio* $p(x)$ *sobre* $GF(2)$ *de grau* n *é dito irredutível se* $p(x)$ *não for divisível por nenhum polinômio sobre* $GF(2)$ *de grau menor que* n *e maior que zero* [1].

Para realizar a extensão do corpo $GF(2^n)$ é preciso escolher um polinômio mônico e irredutível $p(x)$ de grau n em $GF(2)$. O conjunto de 2^n polinômios de grau menor que n formado é chamado de F. A adição e a multiplicação destes polinômios são tomados módulo $p(x)$ e o conjunto F forma um corpo com 2^n elementos, denotado por $GF(2^n)$. $GF(2^n)$ é uma extensão de $GF(2)$[13]. Como exemplo e de forma análoga temos a formação dos números complexos a partir dos números reais. No Quadro 2.2 podemos observar os diferentes tipos de representações para os elementos de um corpo em $GF(2)$, dados pelo conjunto $\{0, 1, x, 1 + x\}$.

Quadro 2.2 - Diferentes representações de elementos de um corpo em $GF(2)$

Notação Inteira	Notação Binária	Notação Polinomial	Notação Exponencial
0	00	0	0
1	01	1	x^0
2	10	x	x^1
3	11	$x+1$	x^2

Se $p(x)$ é um polinômio primitivo de grau n sobre $GF(2)$, então ele divide $x^m + 1$, em que $m = 2^n - 1$ [1]. No Quadro 2.3 pode ser observada uma lista com polinômios primitivos sobre $GF(2)$.

Quadro 2.3 - Polinômios primitivos sobre $GF(2)$

Grau n	Polinômio Primitivo
3	$1 + x^1 + x^3$
4	$1 + x^1 + x^4$
5	$1 + x^2 + x^5$
6	$1 + x^1 + x^6$
7	$1 + x^1 + x^7$
8	$1 + x^2 + x^3 + x^4 + x^8$
9	$1 + x^4 + x^9$
10	$1 + x^3 + x^{10}$
11	$1 + x^2 + x^{11}$
12	$1 + x^1 + x^4 + x^6 + x^{12}$
13	$1 + x^1 + x^3 + x^4 + x^{13}$
14	$1 + x^1 + x^6 + x^{10} + x^{14}$
15	$1 + x^1 + x^{15}$
16	$1 + x^1 + x^3 + x^{12} + x^{16}$

Teorema 2.3 *Para cada elemento não nulo* $\alpha \in GF(q)$ $ord(\alpha)$ *divide* $q - 1$ [1].

Utilizando o algoritmo de Euclides para divisão, temos:

$$q-1=at+r. \tag{2.8}$$
$$\alpha^{q-1}=\alpha^{at+r}. \tag{2.9}$$
$$\alpha^{q-1}=\alpha^{at}\alpha^{r}. \tag{2.10}$$

Como $\alpha^t = 1\ e\ \alpha^{q-1} = 1$, então:

$$p(x)=q(x)d(x)+r(x). \tag{2.11}$$

Assim, $\alpha^r = 1$ e $r = 0$.

Teorema 2.4 *O grupo de elementos não nulos diferentes de GF(q) é um grupo cíclico sobre a multiplicação* [18].

Seja $GF(q)$ um corpo e $GF(p^n)$ a sua extensão. Seja β elemento de $GF(p^n)$. O polinômio f de menor grau tal que $f(\beta) = 0$ é chamado de *polinômio mínimo* de β. Este polinômio existe, é único, irredutível e é utilizado para gerar palavras-código, pois a palavra $P(x)$ só é código se e somente se $P(\beta_1) = 0$.

Para $\alpha \in GF(q)$, seja t o menor inteiro tal que $\alpha^{p^t} = \alpha$. Define-se conjugado de α o conjunto $C = \{\alpha, \alpha^{p^1}, \alpha^{p^2}, \alpha^{p^3}, \alpha^{p^4}, \alpha^{p^5} \ldots, \alpha^{p^t}\}$.

Para representar os 2^n elementos, o conceito de base será introduzido na seção 2.1.3. A escolha da representação a ser utilizada em um corpo sobre uma base depende da implementação a ser adotada.

2.1.3 Bases de Corpos Finitos

Existem diferentes tipos de bases para representar os elementos de um corpo finito. As bases mais utilizadas para projetar operadores aritméticos sobre corpos finitos são a base polinomial, também conhecida como base padrão ou canônica, base normal e base dual. Neste trabalho utiliza-se a base polinomial, pois o multiplicador proposto é baseado no multiplicador de Mastrovito e este é um multiplicador em $GF(2^n)$ de base polinomial.

Um corpo extensão $\mathrm{GF}(2^n)$ de um corpo GF(q) pode ser visto como um espaço vetorial de dimensão n sobre $GF(q)$. Cada elemento de $GF(2^n)$ pode ser representado pela combinação linear dos n elementos de uma base, em que os coeficientes da combinação linear são elementos de $GF(q)$. Em outras palavras, um conjunto de n

elementos linearmente independentes $x = \{x_0, x_1, x_2, x_3, \dots, x_{n-1}\}$ em $GF(2^n)$ é chamado de base de $GF(2^n)$ se um elemento $x \epsilon\ GF(2^n)$ for representado unicamente como uma soma ponderada desta base sobre $GF(2)$.

Definição 2.5 (Base Polinomial) *O conjunto* $\{1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{n-1}\}$*, em que α é uma raiz do polinômio irredutível* $p(x)$ *de grau* n *em* $GF(q)$*, é chamado base polinomial, canônica ou padrão* [17].

A base polinomial é bastante intuitiva, pois está relacionada diretamente à representação dos elementos do corpo de extensão como polinômios definidos em um subcorpo. Um elemento que pertença a $GF(2^n)$ é representado pelo polinômio $A(x) = \alpha_0 + \alpha_1 x^1 + \alpha_2 x^2 + \cdots + \alpha_{n-1} x^{n-1}$, definido em $GF(2)$, e cada elemento de $GF(2^n)$ representa uma classe de resíduo módulo $p(x)$. A representação polinomial $A(x)$ é equivalente a $A(\beta) = \alpha_0 + \alpha_1 \beta^1 + \alpha_2 \beta^2 + \cdots + \alpha_{n-1} \beta^{n-1}$, sendo β uma raiz de $p(x)$.

Uma vantagem de se utilizar a base polinomial em multiplicadores é que os mesmos não requerem conversão de base. Esta característica torna o projeto de multiplicadores em $GF(2^n)$ mais simples e eficientes quando comparados com projetos de multiplicadores utilizando outras bases [2].

Considerando $GF(2^8)$ e o polinômio irredutível em $GF(2)$, sendo $p(x) = x^8 + x^4 + x^3 + x^2 + 1$, tomamos α como a raiz de $p(x)$. Então $A = \{1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7\}$ forma a base polinomial. Todos os 256 elementos podem ser representados como $a = a_0 + a_1\alpha^1 + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 + a_5\alpha^5 + a_6\alpha^6 + a_7\alpha^7$, em que $a_i \epsilon\ GF(2)$.

No Quadro 2.4 são mostradas as diferentes formas de representação de 15 dos 256 elementos de $GF(2^8)$.

Quadro 2.4 - Diferentes representações em $GF(2^8)$

Elemento de $GF(2^8)$.	**Representação Polinomial**	**Representação Vetorial**
0	0	(0,0,0,0,0,0,0,1)
α	α	(0,0,0,0,0,0,1,0)
α^2	α^2	(0,0,0,0,0,1,0,0)
α^3	α^3	(0,0,0,0,1,0,0,0)
α^4	α^4	(0,0,0,1,0,0,0,0)
α^5	α^5	(0,0,1,0,0,0,0,0)
α^6	α^6	(0,1,0,0,0,0,0,0)
α^7	α^7	(1,0,0,0,0,0,0,0)
α^8	$\alpha^4 + \alpha^3 + \alpha^2 + 1$	(0,0,0,1,1,1,0,1)
α^9	$\alpha^5 + \alpha^4 + \alpha^3 + \alpha$	(0,0,1,1,1,0,1,0)
α^{10}	$\alpha^6 + \alpha^5 + \alpha^4 + \alpha^2$	(0,1,1,1,0,1,0,0)
α^{11}	$\alpha^7 + \alpha^6 + \alpha^5 + \alpha^3$	(1,1,1,0,1,0,0,0)
α^{12}	$\alpha^7 + \alpha^6 + \alpha^3 + \alpha^2 + 1$	(1,1,0,0,1,1,0,1)
α^{13}	$\alpha^7 + \alpha^2 + \alpha$	(1,0,0,0,0,1,1,0)
α^{14}	$\alpha^4 + \alpha + 1$	(0,0,0,1,0,0,1,1)

2.1.4 Operadores de adição e subtração em $GF(2^n)$

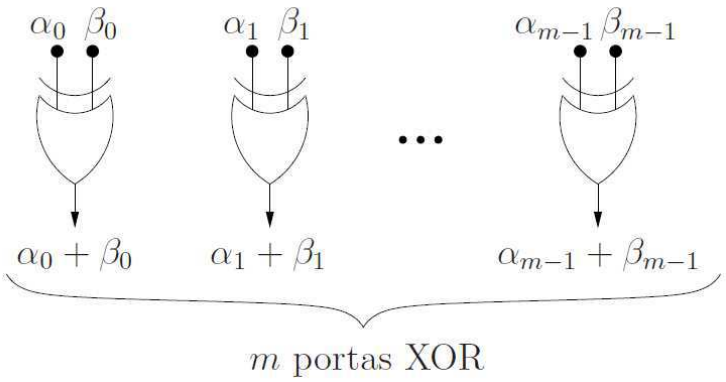
Considere um corpo finito $GF(2^n)$ e uma base $x = \{x_0, x_1, x_2, x_3, \dots, x_{n-1}\}$ para este corpo. A operação de adição de dois elementos $A(x) = \alpha_0 + \alpha_1 x^1 + \alpha_2 x^2 + \cdots + \alpha_{n-1} x^{n-1}$ e $B(x) = \beta_0 + \beta_1 x^1 + \beta_2 x^2 + \cdots + \beta_{n-1} x^{n-1}$ pertencentes a este corpo é realizada adicionando seus coeficientes um a um. Ou seja, $A + B = (\alpha_0 + \beta_0) + (\alpha_1 + \beta_1)x^1 + (\alpha_2 + \beta_2)x^2 + \cdots + (\alpha_{n-1} + \beta_{n-1})x^{n-1}$, sendo $\alpha_i, \beta_i \epsilon\, GF(2)$. Esta operação é representada no Quadro 2.5, na forma de tabela da verdade e na Figura 2.2 na forma de portas lógicas.

Considerando apenas dois elementos, podemos utilizar uma porta lógica XOR para realizar esta operação. Consequentemente, a adição $A + B$ pode ser implementada por um conjunto de m portas XOR.

Quadro 2.5 - Tabela da verdade relativa à operação de adição no corpo $GF(2)$

+	0	1
0	0	1
1	1	0

Figura 2.2 - Operação de adição em $GF(2)$ utilizando portas lógicas XOR [6]



Uma vez que a inversa aditiva de um elemento pertencente a $GF(2^n)$ é o próprio, a operação de subtração em corpos finitos de característica 2 é equivalente à operação de adição [13].

2.1.5 Aritmética Modular com Anéis de Polinômios

Seja um polinômio mônico, coeficiente dominante igual a 1, $d(x)$ de grau m. Todo polinômio $p(x)$ pode ser representado por:

$$p(x) = q(x)d(x) + r(x) \tag{2.12}$$

sendo o grau de $r(x) < m$. Logo, o resto da divisão dos polinômios $p(x)$ e $d(x)$ é denotado por:

$$r(x) = p(x) \bmod d(x) \tag{2.13}$$

O conjunto das possibilidades de restos polinomiais é dado por:

$$R_{p,m=}\left\{r_0+r_1x^1+r_2x^2+\ldots+r_{m-1}x^{m-1}\ |r_j\epsilon P,0\leq\ j\geq m-1\right\} \tag{2.14}$$

As regras e leis da aritmética polinomial são as mesmas para a aritmética modular.

Sendo $r(x)\ =\ p(x)\ mod\ \ d(x)$ e $s(x)\ =\ h(x)\ mod\ d(x)$, ou seja, $r(x)\ =\ p(x)\ -\ q(x)d(x)\ e\ s(x)\ =\ h(x)\ -\ t(x)d(x)$, então:

$$p(x)+h(x)=r(x)+s(x)-\left(q(x)+t(x)\right)d(x) \tag{2.15}$$

$$p(x)h(x)=r(x)s(x)-\left(q(x)s(x)+t(x)r(x)\right)d(x)+q(x)t(x)d^2(x) \tag{2.16}$$

Logo,

$$\left(p(x)+h(x)\right)mod\,d(x)=\left(r(x)+s(x)\right)mod\,d(x) \tag{2.17}$$

$$p(x)h(x)mod\,d(x)=r(x)s(x)mod\,d(x) \tag{2.18}$$

Sendo assim, as leis da adição e multiplicação polinomiais $mod\ p(x)$ são definidas pelas equações (2.19) e (2.20).

$$r(x)+s(x)=\left(r(x)+s(x)\right)mod\,d(x) \tag{2.19}$$

$$r(x)\times s(x)=\left(r(x)\times s(x)\right)mod\,d(x) \tag{2.20}$$

em que no lado esquerdo da equação, $r(x)$ e $s(x)$ são elementos do conjunto de restos polinomiais $R_{p,m}$, enquanto que no lado direito da equação, $r(x)$ e $s(x)$ correspondem a polinômios ordinários [13].

Hardware digital opera intrinsicamente em dois estados e com o código binário. Uma extensão do corpo finito em q, fazendo q uma potência de 2, pois o Campo de Galois $GF(2)$ é mapeado diretamente no domínio digital. Uma subclasse é denotada por $GF(2^n)$. Desta forma, um campo finito $GF\,(16)$ ou F_{16} ou $GF(2^4)$ pode ser representado pelo conjunto de polinômios sobre F_2 de grau menor que 4. A representação é da forma descrita na Equação (2.21).

$$F_{16}=GF\left(2^4\right)=\{a_3x^3+a_2x^2+a_1x^1+a_0x^0\ |a_i\epsilon\{0,1\}\} \tag{2.21}$$

Alternativamente, cada polinômio $a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$ é representado pelo correspondente vetor de coeficientes (a_3, a_2, a_1, a_0) de dimensão 4, conforme a equação (2.22).

$$F_{16} = GF\left(2^4\right) = \{(a_3, a_2, a_1, a_0) \mid a_i \in \{0,1\}\} \tag{2.22}$$

Esta última representação é muito adequada para programas computacionais, já que o polinômio $x^4 + x + 1$ é irredutível para Z_2. Ou seja, $p(x) = x^4 + x + 1$, da forma geral $p(x) = x^n + x + 1$, é um polinômio primitivo em $GF(2^4)$ e com ele pode-se realizar operações neste corpo com módulo de $p(x)$. Uma das operações é a multiplicação.

O primeiro passo para realizar uma multiplicação em $GF(2^4)$ é realizar uma multiplicação usual de polinômios, com a ressalva que os coeficientes sejam (a_3, a_2, a_1, a_0), no qual $a_i \in \{0,1\}$. Então, uma redução de coeficientes utilizando $mod\ 2$ é necessária. O segundo passo é fazer a redução *mod* por um polinômio $p(x) = x^4 + x + 1$ de $GF(2^4)$.

O mesmo procedimento é realizado para se obter a representação de um corpo maior que 4. Por exemplo, a representação em $GF(2^8)$.

$$F_{256} = GF\left(2^8\right) = \{a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 \mid a_i \in \{0,1\}\}. \tag{2.23}$$

Alternativamente, cada polinômio $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$ é representado pelo correspondente vetor de coeficientes $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ de dimensão 8:

$$F_{256} = GF\left(2^8\right) = \{(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) \mid a_i \in \{0,1\}\}. \tag{2.24}$$

2.2 Multiplicação em $GF(2^n)$

Considere p um número primo. O conjunto de restos módulo p, $GF(p) = \{0, 1, \ldots, p-1\}$ é um corpo finito chamado de corpo primo. Todo corpo finito possui um elemento nulo, um elemento de unidade e é uma extensão de grau finito de um corpo primo. Os corpos finitos ou Campo de Galois são na forma $GF(p^n)$, em que p é

primo e n é um número inteiro e positivo. Logo, o corpo $GF(p^n)$ possui p^n elementos [9].

Para um corpo com característica 2, temos $p = 2$, $GF(2^n)$ contem 2^n elementos e seu corpo primo é $GF(2) = \{0,1\}$. O seu grupo multiplicativo possui $2^n - 1$ elementos e um elemento α, elemento primitivo, está contido nestes elementos tal que todos os elementos de $GF(2^n) - \{0\}$ podem ser representados como potências de tal elemento. Assim, tem-se:

$$GF\left(2^n\right) = \{0\} \cup \left\{\alpha^1, \alpha^2, \ldots, \alpha^{2n-2}, \alpha^{2n-1} = 1\right\}. \tag{2.25}$$

Fazendo $P(\alpha) = 0$, tem-se:

$$\alpha^n = Q(\alpha) = q_{n-1}\alpha^{n-1} + q_{n-2}\alpha^{n-2} + \ldots + q_1\alpha^1 + q_0. \tag{2.26}$$

A equação (2.26) é utilizada para representar todos os elementos do corpo como uma combinação linear única das potências $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{n-1}\}$. Logo, qualquer elemento de $GF(2^n)$ pode ser representado na forma polinomial em α e grau n. Ou seja, qualquer elemento pertencente a $GF(2^n)$ pode ser representado da seguinte forma:

$$A(\alpha) = \sum_{i=0}^{n-1} a_i \alpha^i. \tag{2.27}$$

Um vetor binário com n elementos pode ser utilizado para implementação de operações aritméticas em hardware, pois a representação polinomial pode ser realizada na forma vetorial $(a_{n-1}, a_{n-2}, \ldots, a_0)$. Os sinais lógicos "0" e "1" podem ser utilizados para representar os elementos do campo, pois $p = 2$. Assim, a operação de multiplicação em $GF(2^n)$ pode ser implementada utilizando portas lógicas AND.

2.2.1 Multiplicador de Mastrovito

Em [19], *Mastrovito* desenvolve um multiplicador em $GF(2^n)$, paralelo de base polinomial. Sua arquitetura apresenta baixa complexidade temporal e espacial em comparação com outros multiplicadores de base Dual, Normal, Polinomial e de Paar [2,9,19]. O multiplicador de Mastrovito e de base Polinomial possuem complexidade muito próximas, mas o multiplicador Polinomial possui um retardo maior. O multiplicador de Paar possui uma complexidade espacial menor que o de Mastrovito para alguns n específicos, como em $GF\ (2^8)$. Mas para outros corpos sua complexidade

é maior do que o Mastrovito [9]. Uma comparação do retardo máximo e do número de portas AND e XOR necessárias para estes multiplicadores está apresentada no Quadro 2.6.

Quadro 2.6 - Complexidade espacial e temporal de multiplicadores de base polinomial

n	Dual			Normal			Polinomial			Mastrovito			Paar		
	AND	XOR	Retardo	AND	XOR	Retardo	AND	XOR	Retardo	AND	XOR	Retardo	AND	XOR	Retardo
4	16	15	3	28	24	3	16	15	4	16	15	3	12	18	4
6	36	35	4	66	60	4	36	35	6	36	35	4	27	37	5
8	64	77	7	160	160	5	64	77	11	64	90	5	48	62	5
10	100	99	6	180	180	5	100	99	12	100	99	6	75	95	7

Assim, o multiplicador de Mastrovito possui as melhores características dentre os multiplicadores paralelos revistos. Sem ter atraso fixo este multiplicador requer $3n^2 - 2$ portas, cerca de metade do multiplicador de Massey-Omura [20].

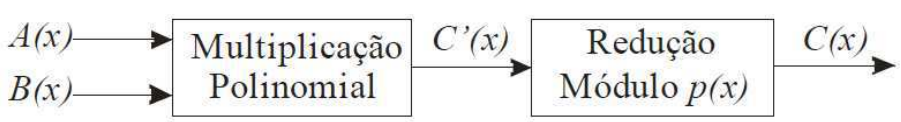
Para o cálculo do produto de dois elementos arbitrários, considere $A(x), B(x)\ e\ C(x) \in GF\ (2^n)$, com $GF\ (2^n)$ escrito na forma polinomial. A multiplicação é realizada da seguinte forma:

$$C(x)=\left[A(x).B(x)\right] mod\ P(x) \tag{2.28}$$

$$C(x)=\left[(a_{n-1}x^{n-1}+\ldots+a_0)\left(b_{n-1}x^{n-1}+\ldots+b_0\right)\right] mod\ P(x) \tag{2.29}$$

em que $P(x)$ é o polinômio irredutível de grau n em $GF(2)$. Este polinômio tem que ser o melhor possível, ótimo, no que diz respeito ao número de portas que se exige para a multiplicação no campo. São utilizados polinômios na forma $P(x) = x^n + x + 1$, para $n = 2,3,4,6,7,9,10,11,15$ e $p(x) = x^n + x^{n-1} + \cdots + x + 1$ [20-21]. Mais informações sobre polinômios irredutíveis são encontradas em [22]. A realização da multiplicação é realizada em duas etapas, como na Figura 2.3.

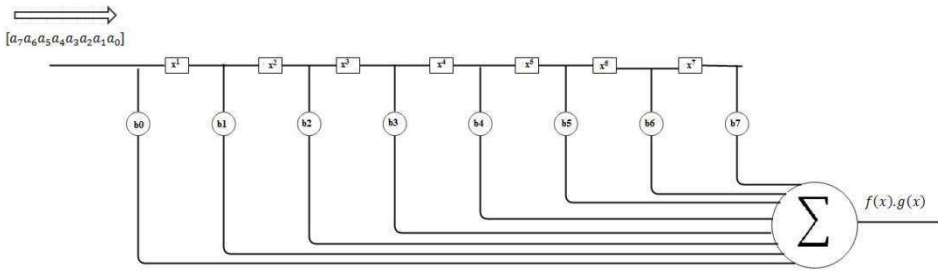
Figura 2.3 - Diagrama de blocos do multiplicador de *Mastrovito*



Na etapa inicial é feita uma multiplicação polinomial, $A(x).B(x)$, como pode-se observar na convolução da Figura 2.4 e Equação (2.30).

$$C'(x) = A(x).B(x) = \left(a_{n-1}x^{n-1} + \ldots + a_0\right)\left(b_{n-1}x^{n-1} + \ldots + b_0\right)$$
$$= a_0b_0 + \left(a_1b_0 + a_0b_1\right)x^1 + \left(a_2b_0 + a_1b_1 + a_0b_2\right)x^2 + \ldots + \left(a_{n-1}b_{n-1}\right)x^{2n-2} \quad (2.30)$$

Figura 2.4 - Multiplicação Polinomial $A(x).B(x)$



Pode-se mostrar o resultado $C'(x)$ da multiplicação como segue na equação (2.31).

$$C'(x) = c'_0 + c'_1 x^1 + c'_2 x^2 + \ldots + c'_{2n-2} x^{2n-1}, \quad (2.31)$$

em que:

$$\begin{aligned} c'_0 &= a_0b_0 \\ c'_1 &= a_1b_0 + a_0b_1 \\ c'_2 &= a_2b_0 + a_1b_1 + a_0b_2 \\ &\cdot \\ &\cdot \\ &\cdot \\ c'_{2n-2} &= a_{n-1}b_{n-1}. \end{aligned} \quad (2.32)$$

Utilizando a forma algébrica, tem-se (2.32):

$$c'_j = \begin{cases} \sum_{i=0}^{j} a_i b_{j-1} & se \quad j < n \\ \sum_{i=j+1-n}^{n-1} a_i b_{j-1} & se \quad n \le j \le 2n-2 \end{cases} \quad (2.33)$$

Esta etapa requer n^2 portas AND independentemente do polinômio escolhido e $n^2 - 2n + 1$ portas XOR.

Para implementar a função paridade com n variáveis e *fan-in* ilimitado são necessárias $2^{n-1} + 1$ portas AON e retardo constante igual a $2\tau_{AON}$, em que τ_{AON} corresponde ao retardo de uma porta lógica tradicional [7]. Seja #AON o número de portas AON. Como cada c'_j é obtido a partir de uma soma de elementos, então o número de portas necessárias para obter os coeficientes é:

$$\begin{aligned} \#AON &= \sum_{j=1}^{n-1}(2^j+1) + \sum_{j=n}^{2n-3}(2^{2n-j-2}+1) \\ &= (n-1) + \sum_{j=1}^{n-1}2^j + (n-2) + 2^{2n-2}\sum_{j=n}^{2n-3}2^{-j} \\ &= (2n-3) + 2^n - 2 + 2^{2n-2}\left(\frac{2^{2n-2}-1}{2^{2n-3}} - \frac{2^n-1}{2^{n-1}}\right) \\ &= 2n-5-2^n+2^{2n-1}-2-2^{2n-1}+2^{n-1} \\ &= 2n-7+2^{n-1}(2+1) \\ &= 3.2^{n-1}+2n-7 \end{aligned} \tag{2.34}$$

Considerando a operação AND bit a bit inicial, que necessidade de n^2 portas, o número total de portas para executar a multiplicação polinomial convencional é:

$$\#AON = 3.2^{n-1} + n^2 + 2n - 7. \tag{2.35}$$

O retardo da multiplicação polinomial é de τ_{AON} para o AND da multiplicação bit a bit e $2\tau_{AON}$ para as soma dos coeficientes.

A implementação da primeira etapa de um multiplicador em $GF(2^4)$, com $p(x) = x^4 + x + 1$, é realizada da seguinte forma:

$$\begin{aligned} C'(x) &= A(x).B(x) \\ C'(x) &= (a_0 + a_1x + a_2x^2 + a_3x^3).(b_0 + b_1x + b_2x^2 + b_3x^3) \\ C'(x) &= a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + \\ &\quad + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 + \\ &\quad + (a_2b_3 + a_3b_2)x^5 + a_3b_3x^6 \end{aligned} \tag{2.36}$$

Assim, cada elemento c'_j de $C'(x)$ é dado por:

$$
\begin{aligned}
c'_0 &= a_0 b_0 \\
c'_1 &= a_0 b_1 + a_1 b_0 \\
c'_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
c'_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
c'_4 &= a_1 b_3 + a_2 b_2 + a_3 b_1 \\
c'_5 &= a_2 b_3 + a_3 b_2 \\
c'_6 &= a_3 b_3
\end{aligned}
\tag{2.37}
$$

Para calcular a multiplicação polinomial convencional de dois elementos em $GF(2^4)$ são necessárias 16 portas que implementem as funções AND e 25 portas que realizem a operação de soma. Esta etapa da multiplicação possui um retardo de $3\tau_{AON}$.

A implementação da primeira etapa de um multiplicador em $GF(2^8)$, com $p(x) = x^8 + x^4 + x^3 + x^2 + 1$, é realizada da seguinte forma:

$$
\begin{aligned}
C'(x) &= A(x).B(x) \\
C'(x) &= (a_0 x^0 + a_1 x^1 + \ldots + a_6 x^6 + a_7 x^7).(b_0 x^0 + b_1 x^1 + \ldots + b_6 x^6 + b_7 x^7) \\
C'(x) &= (a_0 b_0) x^0 + (a_0 b_1 + a_1 b_0) x^1 + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 + \\
&\quad + (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) x^3 + (a_0 b_4 + a_1 b_3 + a_2 b_2 + a_3 b_1 + a_4 b_0) x^4 + \\
&\quad + (a_0 b_5 + a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1 + a_5 b_0) x^5 + \\
&\quad + (a_0 b_6 + a_1 b_5 + a_2 b_4 + a_3 b_3 + a_4 b_2 + a_5 b_1 + a_6 b_0) x^6 + \\
&\quad + (a_0 b_7 + a_1 b_6 + a_2 b_5 + a_3 b_4 + a_4 b_3 + a_5 b_2 + a_6 b_1 + a_7 b_0) x^7 + \\
&\quad + (a_1 b_7 + a_2 b_6 + a_4 b_4 + a_5 b_3 + a_6 b_2 + a_7 b_1) x^8 + \\
&\quad + (a_2 b_7 + a_3 b_6 + a_4 b_5 + a_5 b_4 + a_6 b_3 + a_7 b_2) x^9 + \\
&\quad + (a_3 b_7 + a_4 b_6 + a_5 b_5 + a_6 b_4 + a_7 b_3) x^{10} + \\
&\quad + (a_4 b_7 + a_5 b_6 + a_6 b_5 + a_7 b_4) x^{11} + (a_5 b_7 + a_6 b_6 + a_7 b_5) x^{12} + \\
&\quad + (a_6 b_{7+} a_7 b_6) x^{13} + (a_7 b_7) x^{14}
\end{aligned}
\tag{2.38}
$$

Assim, cada elemento de c'_j de $C'(x)$ é dado por:

$$
\begin{aligned}
c'_0 &= a_0b_0 \\
c'_1 &= a_0b_1 + a_1b_0 \\
c'_2 &= a_0b_2 + a_1b_1 + a_2b_0 \\
c'_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 \\
c'_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 \\
c'_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0 \\
c'_6 &= a_0b_6 + a_1b_5 + a_2b_4 + a_3b_3 + a_4b_2 + a_5b_1 + a_6b_0 \\
c'_7 &= a_0b_7 + a_1b_6 + a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2 + a_6b_1 + a_7b_0 \\
c'_8 &= a_1b_7 + a_2b_6 + a_3b_5 + a_4b_4 + a_5b_3 + a_6b_2 + a_7b_1 \\
c'_9 &= a_2b_7 + a_3b_6 + a_4b_5 + a_5b_4 + a_6b_3 + a_7b_2 \\
c'_{10} &= a_3b_7 + a_4b_6 + a_5b_5 + a_6b_4 + a_7b_3 \\
c'_{11} &= a_4b_7 + a_5b_6 + a_6b_5 + a_7b_4 \\
c'_{12} &= a_5b_7 + a_6b_6 + a_7b_5 \\
c'_{13} &= a_6b_{7+} a_7b_6 \\
c'_{14} &= a_7b_7
\end{aligned}
\tag{2.39}
$$

Para calcular a multiplicação polinomial convencional de 2 elementos em $GF(2^8)$ são necessárias 64 portas que implementem a função AND, multiplicação $a_i b_k$, e 393 portas para implementar as somas.

Na segura etapa é feita uma redução de módulo, que depende do polinômio irredutível escolhido, para encontrar o resultado final $C(x)$, pertencente a $GF\ (2^n)$. Para esta redução são usadas $2n - 2$ portas XOR [7] e a expressão final para $C(x)$ é mostrada na equação (2.40).

$$
\begin{aligned}
C(x) = &\left(c'_0 + c'_n\right) + \left(c'_1 + c'_n + c'_{n+1}\right)x^1 + \ldots + \\
&+ \left(c'_{n-2} + c'_{2n-2} + c'_{2n-3}\right)x^{n-2} + \left(c'_{n-1} + c'_{2n-2}\right)x^{n-1},
\end{aligned}
\tag{2.40}
$$

Novamente, podemos rescreve-se a equação (2.40):

$$
C(x) = c_0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1},
\tag{2.41}
$$

em que:

$$
\begin{aligned}
c_0 &= c'_0 + c'_n \\
c_1 &= c'_1 + c'_n + c'_{n+1} \\
&\cdot \\
&\cdot \\
&\cdot \\
c_{n-1} &= c'_{n-1} + c'_{2n-2}\,.
\end{aligned}
\tag{2.42}
$$

Na forma algébrica, tem-se:

$$\begin{aligned} c_0 &= c'_0 + c'_n \\ c_{n-1} &= c'_{n+1} + c'_{2n-2} \\ &\vdots \\ c_j &= c'_j + c'_{j+n+1} + \ldots + c'_{j+n}, \end{aligned} \tag{2.43}$$

em que $j = 1,2,3,\ldots,n-2$.

Calculando o número de portas para gerar cada termo c_i da equação, tem-se:

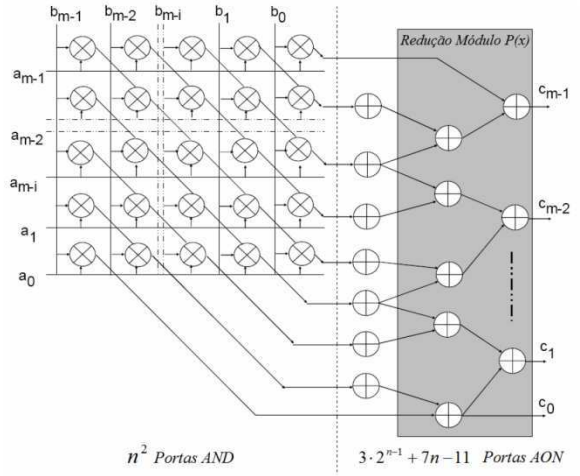
$$\begin{aligned} \#AON &= 3 + \sum_{j=1}^{n-2} 5 + 3 \\ \#AON &= 6 + 5(n-2) \\ \#AON &= 5n - 4 \end{aligned} \tag{2.44}$$

O total de portas do multiplicador em $GF(2^n)$, considerando a multiplicação polinomial e a redução de módulo, é dado por:

$$\#AON = 3.2^{n-1} + n^2 + 7n - 11 \tag{2.45}$$

Para a multiplicação em $GF\,(2^4)$ e $GF\,(2^8)$ são necessárias 16 e 64 portas lógicas AND, respectivamente, para a primeira etapa. Para a realização da segunda etapa, redução de módulo, o multiplicador em $GF\,(2^4)$ e $GF\,(2^8)$ necessita de 41 e 429 portas, respectivamente. No total, o multiplicador em $GF\,(2^4)$ realiza a operação com 57 portas AON e o multiplicador em $GF\,(2^8)$ com 493 portas AON. Na Figura 2.5 é mostrado um diagrama de blocos do processo de multiplicação em corpos finitos em *hardware*.

Figura 2.5 - Diagrama da implementação em *hardware* de multiplicadores em $GF\,(2^n)$



Na implementação de multiplicadores utilizando a lógica tradicional geralmente é encontrado, na literatura, circuitos com *fan-in* limitado. Este procedimento é realizado para diminuir o tamanho do circuito, visto que a complexidade do circuito para o multiplicador de Mastrovito é exponencial e dado por $3.2^{n-1} + n^2 + 7n - 11$ [1]. Com um *fan-in* ilimitado é possível reduzir o número de portas de um crescimento exponencial para um crescimento polinomial. Assim, é possível obter uma arquitetura do multiplicador de Mastrovito que forneça um número menor de portas, mas com a desvantagem de o atraso não ser fixo [20].

Cada caminho através do multiplicador possui apenas uma porta AND e, no máximo, $2\log n$ portas XOR. Levando em consideração todas as portas e o atraso de cada uma delas, o atraso total é dado pela equação (2.46).

$$T \leq \tau_{AND} + 2\tau_{XOR}\left[\log_2 n\right] \tag{2.46}$$

em que τ_{AND} e τ_{XOR} são os atrasos correspondentes as duas portas lógicas. Dessa forma, somando os retardos produzidos por cada camada do multiplicador é obtido um retardo total de $5\tau_{AND}$, τ_{AND} para o AND bit a bit, $2\tau_{AND}$para a multiplicação polinomial e $2\tau_{AND}$para a redução módulo $p(x)$. τ_{AND} é o tempo de atraso correspondente a uma porta AON.

A multiplicação 15 com 15 em decimal, por exemplo, pode-se fazê-la sob diferentes representações. Logo, pode-se escrever:

$$15_{10} \times 15_{10} = F_{16} \times F_{16} = 1111_2 \times 1111_2 = (x^3 + x^2 + x^1 + x^0) \times (x^3 + x^2\, x^1 + x^0) \tag{2.47}$$

Aplicando o primeiro passo, multiplicação e redução por *mod2,* tem-se:

$$(x^3 + x^2 + x^1 + x^0) \times (x^3 + x^2 + x^1 + x^0) = x^6 + 2x^5 + 3x^4 + 4x^3 + 3x^2 + 2x + 1 \tag{2.48}$$

$$(x^3 + x^2 + x^1 + x^0) \times (x^3 + x^2 + x^1 + x^0) = x^6 + x^4 + x^2 + 1 \tag{2.49}$$

Aplicando o segundo passo, redução *mod* por um polinômio primitivo de $GF(2^4)$, tem-se:

$$(x^6 + x^4 + x^2 + 1)\ mod\ p(x) = (x^6 + x^4 + x^2 + 1)\ mod\ (x^4 + x + 1) = -x^3 - x^1 = x^3 + x^1 \tag{2.50}$$

Portanto, esta multiplicação requer 57 portas tradicionais. O resultado pode ser exposto nas diferentes representações:

$$10_{10} = A_{16} = 1010_2 = x^3 + 0x^2 + x^1 + 0 = x^3 + x^1 \qquad (2.51)$$

No Quadro 2.7 e 2.8, podem ser visualizados os resultados, em hexadecimal e decimal, para a multiplicação dos elementos do $GF(2^4)$, aplicando os passos descritos nesta seção.

Quadro 2.7 - Resultado da multiplicação em $GF(2^4)$ em hexadecimal

×	2	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	3	1	7	5	B	9	F	D
3	0	3	6	5	C	F	A	9	B	8	D	E	7	4	1	2
4	0	4	8	C	3	7	B	F	6	2	E	A	5	1	D	9
5	0	5	A	F	7	2	D	8	E	B	4	1	9	C	3	6
6	0	6	C	A	B	D	7	1	5	3	9	F	E	8	2	4
7	0	7	E	9	F	8	1	6	D	A	3	4	2	5	C	B
8	0	8	3	B	6	E	5	D	C	4	F	7	A	2	9	1
9	0	9	1	8	2	B	3	A	4	D	5	C	6	F	7	E
A	0	A	7	D	E	4	9	3	F	5	8	2	1	B	6	C
B	0	B	5	3	A	1	F	4	7	C	2	9	D	6	8	3
C	0	C	B	7	5	9	E	2	A	6	1	D	F	3	4	8
D	0	D	9	4	1	C	8	5	2	F	B	6	3	E	A	7
E	0	E	F	1	D	3	2	C	9	7	6	8	4	A	B	5
F	0	F	D	2	9	6	4	B	1	E	C	3	8	7	5	A

Quadro 2.8 - Resultado da multiplicação em $GF(2^4)$ em decimal

×	2	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	4	6	8	10	12	14	3	1	7	5	11	9	15	13
3	0	3	6	5	12	15	10	9	11	8	13	14	7	4	1	2
4	0	4	8	12	3	7	11	15	6	2	14	10	5	1	13	9
5	0	5	10	15	7	2	13	8	14	11	4	1	9	12	3	6
6	0	6	12	10	11	13	7	1	5	3	9	15	14	8	2	4
7	0	7	14	9	15	8	1	6	13	10	3	4	2	5	12	11
8	0	8	3	11	6	14	5	13	12	4	15	7	10	2	9	1
9	0	9	1	8	2	11	3	10	4	13	5	12	6	15	7	14
A	0	10	7	13	14	4	9	3	15	5	8	2	1	11	6	12
B	0	11	5	3	10	1	15	4	7	12	2	9	13	6	8	3
C	0	12	11	7	5	9	14	2	10	6	1	13	15	3	4	8
D	0	13	9	4	1	12	8	5	2	15	11	6	3	14	10	7
E	0	14	15	1	13	3	2	12	9	7	6	8	4	10	11	5
F	0	15	13	2	9	6	4	11	1	14	12	3	8	7	5	10

Aplicando o mesmo procedimento algébrico para $GF(2^8)$ e utilizando uma função do Matlab, temos a tábua de multiplicação para esta extensão de $GF(2)$. A função $primpoly('n', all)$ fornece todos os polinômios primitivos para qualquer n. Especificamente para $n = 8$, tem-se 16 polinômios gerados.

```
>> primpoly(8,'all')
Primitive polynomial(s) =
D^8+D^4+D^3+D^2+1
D^8+D^5+D^3+D^1+1
D^8+D^5+D^3+D^2+1
D^8+D^6+D^3+D^2+1
D^8+D^6+D^5+D^1+1
D^8+D^6+D^5+D^2+1
D^8+D^6+D^5+D^3+1
D^8+D^6+D^5+D^4+1
D^8+D^7+D^2+D^1+1
D^8+D^7+D^3+D^2+1
D^8+D^7+D^5+D^3+1
D^8+D^7+D^6+D^1+1
D^8+D^7+D^6+D^3+D^2+D^1+1
```

D^8+D^7+D^6+D^5+D^2+D^1+1
D^8+D^7+D^6+D^5+D^4+D^2+1
D^8+D^6+D^4+D^3+D^2+D^1+1

Utilizando o polinômio primitivo $D^8 + D^4 + D^3 + D^2 + 1$ no algoritmo descrito a seguir, gera-se a tábua de multiplicação para o $GF(2^8)$. Para cada polinômio primitivo diferente utilizado é projetado um circuito diferente que realiza a mesma operação. Este algoritmo computa todas as tábuas para qualquer tamanho de corpo a depender da escolha do da variável n.

```
n = 8;
x = gf([0:2^n-1]',n);
multb = x * x'
multb = GF(2^8)
%polinomio_primitivo = D^8+D^4+D^3+D^2+1 (100011101b, 28510 e 11DHEX)
```

O resultado é uma matriz simétrica com 256 linhas e 256 colunas e 65536 elementos, disponível no Anexo 1.

2.3 Conclusão

Neste capítulo foi realizada uma revisão bibliográfica da aritmética em corpos finitos e suas propriedades, bem como a definição dos operadores de adição, subtração e multiplicação neste campo. Uma breve revisão do Multiplicador de Mastrovito foi realizada e sua escolha foi justificada baseada na comparação das complexidades espaciais e temporais com outras arquiteturas de multiplicadores de base polinomial. Assim, foi projetado o Multiplicador de Mastrovito em $GF(2^4)$ e $GF(2^8)$ e a tábua de multiplicação para os dois casos. Logo, a implementação em *hardware* de operações aritméticas em corpos finitos é uma boa alternativa em relação à implementação via *software* e pode ser realizada projetando e desenvolvendo circuitos VLSI (*Very Large Scale Integration*) ou circuitos programáveis, a exemplo do FPGA.

3. REDES NEURAIS ARTIFICIAIS

3. Redes Neurais Artificiais

O objetivo deste capítulo é introduzir importantes aspectos computacionais de uma rede neural artificial e apresentar o modelo utilizado neste trabalho. Das muitas áreas em que fazem uso das redes neurais, concentra-se no estudo de modelos discretos, em que conjuntos discretos na entrada e saída estão relacionados por meio de parâmetros da rede. Discutem-se alguns aspectos da computação neural, aprendizado e representação, assim como os conceitos em computação neural discreta e complexidade.

Neste capítulo é introduzida a arquitetura das redes neurais discretas utilizadas ao longo desta dissertação e apresentado o circuito de limiar, que desempenha a função de uma porta neural ou neurônio artificial. Na sequência é mostrado como construir circuitos de limiar com 2 e 3 camadas.

3.1 Introdução

O interesse no desenvolvimento de pesquisas sobre redes neurais artificiais surgiu da convicção que o cérebro humano é bastante superior aos computadores em resolver diversos tipos de problemas, tais como reconhecimento de voz e imagens. Estas redes neurais podem ser melhores descritas com redes de alto grau de interconexão com várias unidades básicas chamadas de portas neurais, inspiradas no neurônio biológico [1].

Os modelos podem ser classificados em duas categorias gerais. A primeira categoria consiste em uma Rede de Propagação Direta, tais como circuitos de limiar e suas variantes. A segunda categoria utiliza conexões de realimentação, como o modelo *Hopfield* de tempo discreto. Os parâmetros de uma rede neural residem nas forças de conexão entre os neurônios. No modelo de propagação direta, os neurônios podem ser arranjados em camadas em que cada elemento de uma camada computa apenas uma função que depende apenas dos valores da camada anterior. Um circuito de limiar é um exemplo deste tipo de rede. Ou seja, cada elemento em uma camada calcula uma função de limiar que depende apenas dos valores das saídas da camada anterior.

A arquitetura da rede neural discreta é formada por portas de limiar linear, onde as camadas mais próximas da entrada são atualizadas primeiramente. Cada neurônio é um elemento

de limiar linear e um conjunto delas forma o circuito de limiar linear. O cálculo de um circuito de limiar pode ser considerado como um processo de propagação direta: as saídas dos neurônios são atualizadas camada por camada, com a camada mais próxima às entradas sendo atualizada primeiro. As saídas dos circuitos são tomadas como as saídas dos neurônios especificados na última camada. Nesse sentido, o número de camadas corresponde ao tempo de computação paralelo no circuito. Se as entradas são binárias, então o circuito de limiar torna-se um circuito combinacional booleano. Uma variante do modelo de circuito de limiar é uma rede de alimentação de entrada de elementos sigmoidais que computam funções contínuas, sendo a saída uma função degrau, característica dos elementos de limiar lineares.

Outro modelo de redes neurais artificiais dizem respeito à Rede *Hopfield* discreta no tempo. Esta rede compreende a um conjunto de elementos inteiramente ligados aos elementos de limiar lineares, onde a saída de cada elemento depende da saída de todos os outros elementos, tal como determinados pela força de conexão entre eles. Pode ser considerado como um circuito booleano sequencial e, assim, difere da natureza combinatória de computação em um circuito de limiar. As entradas para a rede são aplicadas como estados iniciais. A cada passo da computação em tempo discreto os estados de um subconjunto dos elementos são atualizados. Para escolhas apropriadas dos parâmetros da rede, a rede passa por uma sucessão de estágios até atingir um estado estável, quando as saídas dos elementos permanecerem constantes. O cálculo executado pela rede pode ser visto como um processo de evolução de um sistema dinâmico.

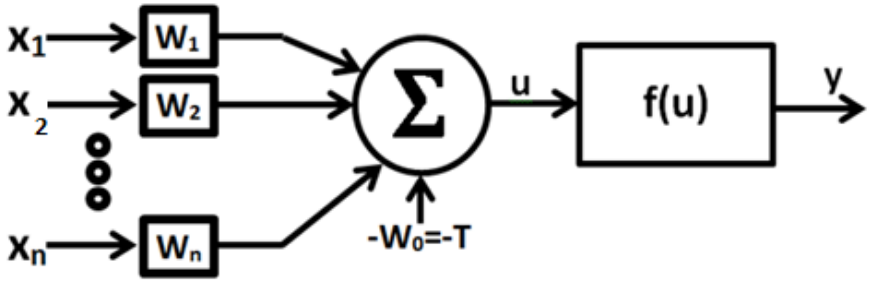
As redes neurais artificiais surgiram, de forma expressiva, pelo trabalho de *McCulloch e Pitts* (1943), em que propuseram um modelo artificial de um neurônio que funcionava de forma discreta e era capaz de realizar várias operações lógicas [10]. Eles propuseram um modelo matemático do neurônio capaz de computar funções antes calculadas com a lógica booleana. A saída desse modelo de neurônio artificial era 1 ou 0, refletindo o estado *tudo-ou-nada* do neurônio biológico. Neste estado, quando o total de entradas alcançasse níveis críticos, o neurônio enviaria sua saída para outros neurônios no qual ele estaria conectado. Embora não seja capaz de capturar toda complexidade de um neurônio biológico, o modelo proposto alia simplicidade com potencial, pois organizando neurônios em uma rede é possível obter estruturas bem mais complexas. No entanto, esta simplicidade é uma vantagem quando o objetivo é

projetar computadores digitais. Um circuito é booleano se as entradas e as saídas deste rede são binárias.

Um aspecto importante das redes neurais é a capacidade de aprendizagem. O mecanismo de aprendizagem pode ser visto como um mecanismo de adaptação de parâmetros, tornando possível que a rede resultante forneça o resultado esperado. A regra que governa como essas mudanças ocorrem é chamada de algoritmo de aprendizagem. O algoritmo de aprendizagem é quem faz o mapeamento da entrada para a saída através de alterações de parâmetros internos da rede. Para isso é necessário assumir uma arquitetura inicial para a rede de forma que seja obtido o mapeamento adequado, ou seja, o algoritmo de aprendizagem irá convergir para um resultado satisfatório. Quando não for possível o término do algoritmo de aprendizagem, o mapeamento desejado não poderá ser representado pela arquitetura utilizada. Portanto, é interessante utilizar uma arquitetura adequada de forma que sempre seja possível a representação do mapeamento. Também é possível indicar uma arquitetura ótima de tal forma que para representar certo mapeamento seja necessário um número mínimo de recursos como, por exemplo, o número de portas neurais. Ao invés de utilizar arquiteturas baseadas em neurônios que de alguma forma aprendem, é possível utilizar arquiteturas cujos neurônios já possuam seus parâmetros previamente calculados [2].

Rosemblett, em 1958, introduziu o conceito do *perceptron*, que mudava a abordagem para o problema de reconhecimento de padrões. Ele desenvolveu um algoritmo para ajuste dos pesos $(W_1, W_2, W_3,...,W_n)$ e mostrou que a resposta convergia quando os padrões são linearmente separáveis por um limiar $-T$ ou $-W_0$ [10]. Na Figura 3.1, é apresentado o modelo proposto por *Rosemblett*, em que $(X_1, X_2, X_3,..., X_n)$ são as entradas, $(W_1, W_2, W_3,...,W_n)$ os pesos, $-T$ ou $-W_0$ o limiar, $f(u)$ a função de ativação e y a saída.

Figura 3.1- Modelo de um neurônio por *Rosemblett*



Por meio de portas de limiar linear é possível utilizar arquiteturas cujos neurônios possuam parâmetros previamente calculados, sem a necessidade de utilizar nenhum algoritmo de aprendizagem. Dessa forma, podem-se projetar diretamente redes neurais discretas e ótimas para representar certos mapeamentos.

Tradicionalmente, o multiplicador de *Mastrovito* tem sido usado como elemento básico na construção de calculadores em corpos finitos. Esta arquitetura normalmente supõe a utilização de portas lógicas elementares AND, OR, NOT na sua implementação. Um dos objetivos do estudo das arquiteturas de processadores é a redução da complexidade espacial e temporal. Estes aspectos das complexidades relacionam-se diretamente com a área do *chip,* quantidade de transistores e com a dissipação de energia. Outro aspecto importante é o retardo do circuito, normalmente avaliado pelo número de camadas de portas elementares existentes na arquitetura [1].

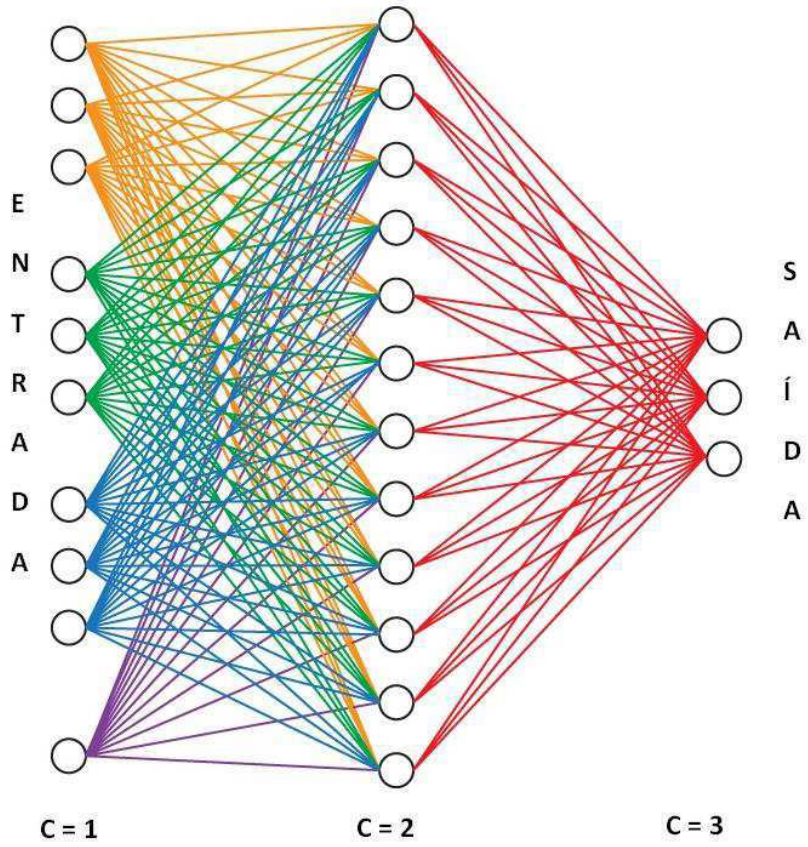
Também está estabelecido na literatura que os neurônios de limiar linear, definidos mais adiante, exibem um poder de computação maior que as portas lógicas tradicionais AND, OR, NOT [7] e cada neurônio é modelado como um elemento de limiar linear com uma saída binária.

Em [1], foi demonstrada a vantagem em utilizar neurônios de limiar linear na construção de um multiplicador de *Mastrovito* para elementos de $GF(2^n)$. Esta implementação permite a redução em cerca de metade do número de portas em relação à implementação com as portas tradicionais. Além dessa vantagem, tem-se que o retardo permanece constante com o aumento do corpo, contrariamente à lógica tradicional.

3.2 Circuito de Limiar Linear

A rede neural utilizada é implementada por um circuito de limiar linear utilizando o Modelo de Propagação Direta. Na Figura 3.2 tem-se a representação de uma rede com três camadas, $C = 1, C = 2$ *e* $C = 3$, sendo a direção da propagação é da esquerda para a direita.

Figura 3.2 - Representação do Modelo de Propagação Direta



O número de portas necessárias para implementar determinada função tem uma relação direta com o tamanho do circuito. Quanto mais portas necessárias, maior o circuito. Como o a propagação é direta, o número de camadas (profundidade da rede) está relacionado com a velocidade de execução. O número de conexões na entrada de uma porta é chamado de *fan-in*. Analogamente, *fan-out* é o número de conexões na saída de uma porta que pode alimentar a entrada de outra, sem degradar o desempenho do circuito. A porta com a maior razão *fan-in/fan-out* determina o *fan-in/fan-out* do circuito. Os circuitos *MOS* podem ser ligados a um número

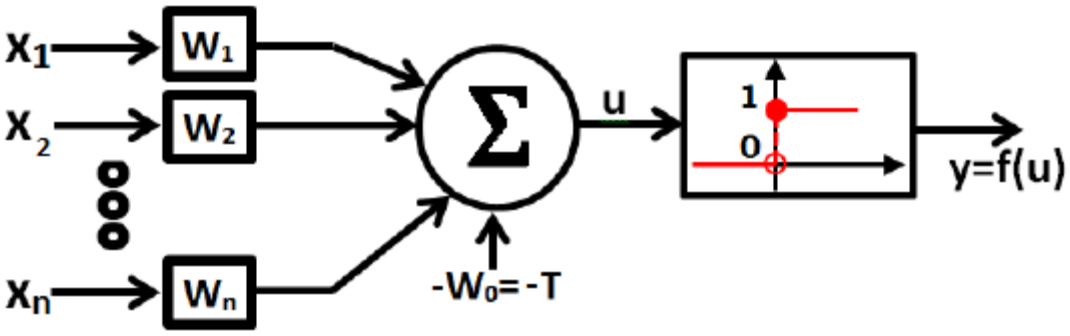
grande de outras portas *MOS* sem degradação do sinal, mas o atraso aumenta com o número de ligações.

3.3 Portas de Limiar Linear

A demanda por alto desempenho de processadores em escala de integração VLSI tem impulsionado as pesquisas por sistemas que garantam alto desempenho utilizando pouca área em chip e que apresentem baixo consumo de potência [2].

As portas lógicas de limiar vêm sendo estudadas desde o decênio de 1960 e algumas implementações eficientes utilizando a tecnologia CMOS são descritas em [23-24]. Existem muitas maneiras de implementar as portas de limiar linear, também chamadas de porta lógica de limiar, em CMOS utilizando *hardware* e *software*. Fatores como velocidade, máxima soma dos pesos, área ocupada e dissipação de energia são levados em consideração. Tais portas podem calcular funções que sejam separáveis linearmente. A sua saída y é função da soma ponderada das entradas e do limiar. Na Figura 3.3, tem-se um modelo de um neurônio artificial que descreve uma porta de limiar linear.

Figura 3.3 - Modelo de uma porta de limiar linear



Da Figura 3.3, tem-se que u é a soma dos sinais de entrada $(X_1, X_2, X_3, \dots, X_n)$ multiplicados pelos respectivos pesos $(W_1, W_2, W_3, \dots, W_n)$ e subtraídos de um nível DC, ou seja, W_0. Logo, tem-se u dado pela expressão (3.1).

$$u = \sum_{i=1}^{n} W_i X_i - W_0 \qquad (3.1)$$

O sinal u é submetido a uma função de ativação $f(u)$, que no caso de uma porta de limiar linear é a função degrau unitário. Para $u \geq 0, f(u) = 1$ e para $u < 0, f(u) = 0$.

Assim, pode-se expressar a saída y evidenciando o limiar de ativação W_0. Logo, a equação (3.1) pode ser expressa como na equação (3.2):

$$y = f(X_1, X_2, X_3, \dots, X_n) = \begin{cases} 1, se \sum_{i=1}^{n} W_i X_i \geq W_0 \\ 0, se \sum_{i=1}^{n} W_i X_i < W_0. \end{cases} \quad (3.2)$$

Desta forma, uma porta de limiar linear realiza a comparação entre o valor do somatório das entradas multiplicadas pelos seus respectivos pesos e subtraídos do valor limiar. Se este valor for maior ou igual a W_0, tem-se uma saída 1. Caso o valor seja menor do que o limiar W_0, tem-se uma saída 0. A porta de limiar linear é a estrutura básica da rede neural discreta e possui a porta tradicional AON como um subconjunto. Logo, elas têm o comportamento, na pior hipótese, igual ao da lógica tradicional [25]. Esta potencialidade das portas pode ser estendida a outros tipos de sistemas numéricos, como o Corpo Finito, amplamente usado em comunicação digital.

Pode ser observada no Quadro 3.1 uma soma aritmética das variáveis binárias de entrada com os pesos unitários pré-estabelecidos. O limiar da função lógica AND é igual a três, $W_0 = 3$, e o limiar da função lógica OR é igual a um, $W_0 = 1$. Desta forma, utilizando a função *sigmoidal*, pode-se escrever as funções lógicas pelas equações 3.3 e 3.4.

$$AND(x_1, x_2, x_3) = sgn(x_1 + x_2 + x_3 - 3). \quad (3.3)$$

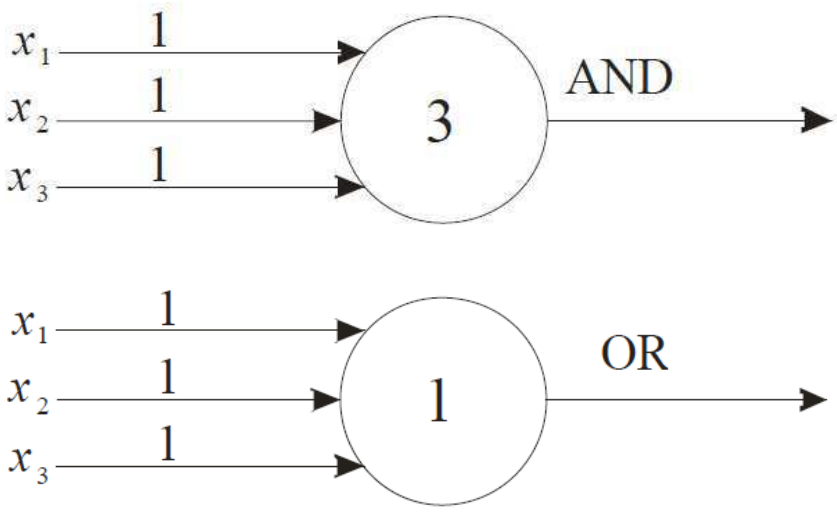
$$OR(x_1, x_2, x_3) = sgn(x_1 + x_2 + x_3 - 1). \quad (3.4)$$

Quadro 3.1 - Limiar das portas AND e OR

x_3	x_2	x_1	$\sum$	AND	OR
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	0	1
1	0	0	1	0	1
1	0	1	2	0	1
1	1	0	2	0	1
1	1	1	3	1	1

O modelo simplificado destas duas portas de limiar é visualizado na Figura 3.4. A partir do modelo da porta de limiar linear, tem-se os pesos unitários nas entradas e os limiares das portas AND e OR. Para a função lógica AND a saída é 1 quando a soma das entradas multiplicadas pelo respectivo peso for igual ou maior do que 3. Caso contrário, 0. Para a função lógica OR, quando a soma das entradas multiplicadas pelo respectivo pelo for maior ou igual a 1, a saída também é nível alto. Caso contrário, nível baixo.

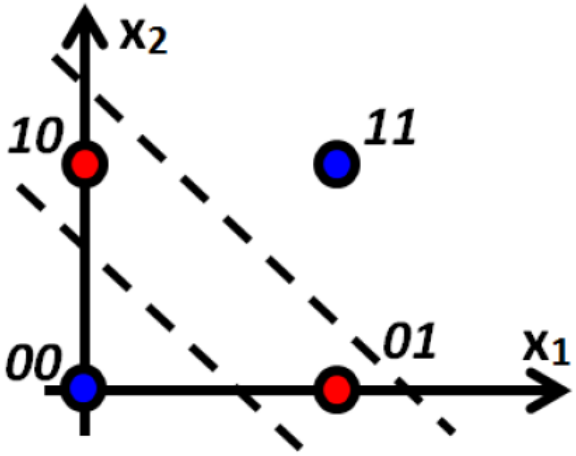
Figura 3.4 - Modelo simplificado das duas portas lógicas



Pelo fato de não ser possível utilizar uma porta de limiar linear para implementar a lógica XOR, ela é um contraexemplo. Com apenas um limiar não é possível limitar os estágios lógicos da saída. Na Figura 3.5, é apresentada uma ilustração gráfica deste fato. As entradas x_1 e x_2 formam um plano com todas as combinações possíveis dos valores de entrada e não é possível, apenas com uma reta, limitar as entradas 01 e 10. A estas entradas corresponderia a saída 1.

Figura 3.5 - Implementação da porta lógica XOR

x_2	x_1	Σ	XOR
0	0	0	0
0	1	1	1
1	0	1	1
1	1	2	0



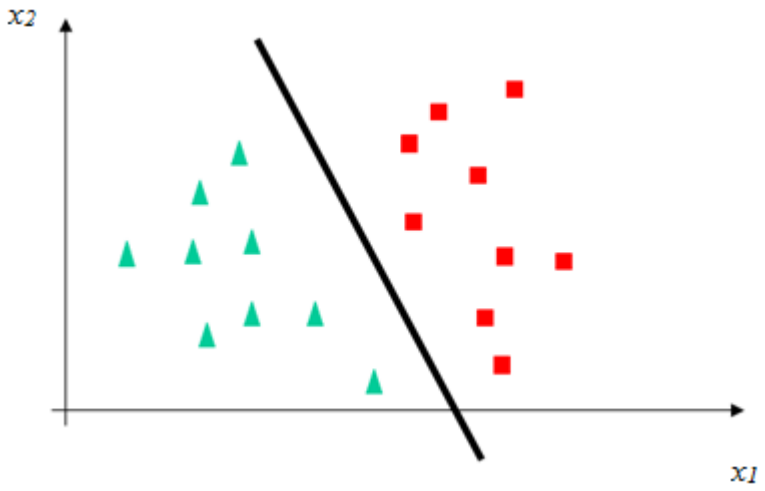
Uma maneira de solucionar este problema é a criação de uma rede neural discreta multicamada. Assim, estas redes são capazes de delimitarem vários outros limiares.

3.4 Funções Simétricas

Uma função booleana f é dita simétrica se sua saída depende apenas do somatório dos seus valores de entrada. Portanto $f(x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n)$, para qualquer permutação de $(x_1, x_2, \dots, x_n)$. A função paridade é um exemplo de função simétrica, cuja saída é 1 se a soma das entradas for ímpar e 0 se a soma das entradas for par [7].

As portas de limiar linear estão restritas às soluções de problemas que sejam linearmente separáveis, ou seja, a problemas cuja solução pode ser obtida pela separação de duas regiões por meio de uma reta (ou um hiperplano para o caso n-dimensional) [1]. A equação de separação linear pode ser facilmente visualizada na Figura 3.6, para o caso bidimensional.

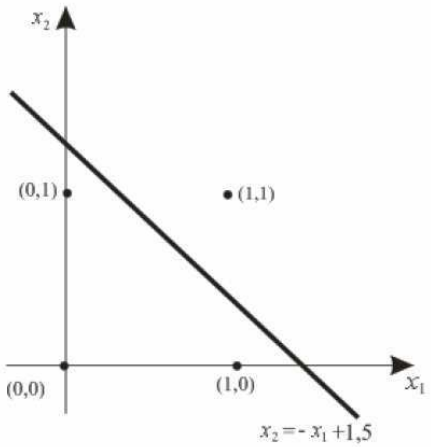
Figura 3.6 - Separação de Regiões por meio de uma reta



Considere, como exemplo ilustrativo, um nó com duas entradas x_1 e x_2, pesos w_1 e w_2 e w_2, limiar t e saída y executando uma função qualquer. A condição de disparo do nó ($y = 1$) é então definida por $x_1 w_1 + x_2 w_2 = t$, que pode ser descrita na forma geral da equação de uma reta em que $x_2 = f(x_1)$, conforme é mostrada na equação 3.5. Portanto, a superfície de decisões de uma porta de limiar linear está restrita a uma reta. Na Figura 3.7 é mostrada a solução para o problema do AND lógico por meio de uma porta de limiar linear.

$$x_2 = -\left(\frac{w_1}{w_2}\right) x_1 + \left(\frac{t}{w_2}\right). \tag{3.5}$$

Figura 3.7- (Solução para o problema do AND lógico utilizando uma porta de limiar linear com w_1=w_2 = 1 e t = 1,5) [2]



Para o caso particular em que $x \epsilon \{0,1\}^n$ que restringe as entradas a valores binários, tem-se que $y : \{0,1\}^n \rightarrow \{0,1\}$ para $w \epsilon R^n$. Apesar de estarem limitadas à resolução de problemas linearmente separáveis, que corresponde a uma pequena parcela do total de 2^{2^n} funções booleanas possíveis com n entradas, as portas de limiar lineares são mais poderosas do que as portas lógicas convencionais. Com uma mesma porta de limiar linear pode-se implementar qualquer uma das funções AND, OR, NAND e NOR, entre outras, bastando mudar os parâmetros da porta [7].

3.5 Construção de Circuitos de 2 e 3 camadas

Projetar circuitos de limiar linear significa implementar funções booleanas. Dentre estas funções é de particular interesse as funções booleanas simétricas. Como as portas de limiar linear estão restritas à solução de funções simétricas, o projeto de circuitos de limiar compreende a implementações de funções deste tipo.

Teorema 3.1. Toda função booleana $f(x_1, x_2, \dots, x_n): \{0,1\} \rightarrow \{0,1\}$ pode ser computada por um circuito de limiar com duas camadas e no máximo $2^{n-1} + 1$ portas de limiar linear [7]

Qualquer função booleana pode ser escrita como uma SDP (Soma de Produtos) como se segue:

$$f(x_1, x_2, \dots, x_n) = P_1 \vee P_2 \vee \dots \vee P_n, \tag{3.6}$$

em que $\vee$ é o símbolo AND, o termo P_i é um produto implementado por esta função de n variáveis e $x' \leq 2^n$.

Analogamente, pode-se escrever a função descrita em (3.6) como um PDS (Produto de Somas) como se segue:

$$f(x_1, x_2, \dots, x_n) = S_1 \wedge S_2 \wedge \dots \wedge S_n, \tag{3.7}$$

em que $\wedge$ é o símbolo OR, cada termo S_i é uma soma implementada por esta função de n variáveis e $x'' \leq 2^n$.

Como $x' + x" = 2^n$, temos que $x' \leq 2^n$ e $x' \leq 2^n$ porque 2^n é o número máximo possível de somas ou de produtos com n variáveis booleanas. A primeira camada do circuito de

limiar irá implementar, para a SDP, os produtos P_i e a segunda camada, utilizando a função OR implementará um PDS, utilizando $2^{n-1}+1$ portas lógicas tradicionais. Como a porta tradicional é um subconjunto de uma porta neural, esta pode simular as portas AND e OR e implementar qualquer função booleana utilizando $2^{n-1}+1$ portas neurais [26]. Nota-se que o crescimento do número de portas com o tamanho do corpo é exponencial. Por este motivo faz-se necessário o estudo de técnicas que utilizem as redes neurais discretas para a construção de circuitos de limiar que computem funções de maneira mais eficiente e que o crescimento do número de portas com o tamanho do corpo seja polinomial. Nas duas subseções a seguir, 3.4.1 e 3.4.2, são apresentadas técnicas para a construção de circuitos de limiar de duas camadas que computam qualquer função booleana simétrica com, no máximo, $n+1$ portas e de três camadas com $2\sqrt{n}+1$ portas, em que n é o número de entradas do circuito

3.5.1 Circuitos de duas camadas

Seja $X=[x_1, x_2, \dots, x_n]$ a entrada booleana do circuito e f uma função booleana simétrica. Como a função f depende somente da soma das entradas, existe um conjunto de valores em $\sum_{k=1}^{n} x_k$ no qual a função é 1. Fazendo um agrupamento destes valores em subintervalos de $[0, n]$, tem-se os subintervalos dados a seguir.

$$[q_1, q'_1], [q_2, q'_2], \dots, [q_s, q'_s], \tag{3.8}$$

em que q_k e q'_ksão inteiros, $q_{k+1} > q'_{k+1}$ e $q_k \leq q'_k$, de tal forma que $f(x_1, x_2, \dots, x_n)=1$, se e somente se para algum j, tal que:

$$\sum_{k=1}^{n} x_k \in [q_j, q'_j]. \tag{3.9}$$

Na primeira camada do circuito, são necessárias 2s portas de limiar linear calculando:

$$y_{q_j} = sgn(\sum_{k=1}^{n} x_k - q_j), \tag{3.10}$$

e

$$y'_{q_j} = sgn(q'_j - \sum_{k=1}^{n} x_k), \tag{3.11}$$

em que $j = 1,2,3,\ldots s$.

A segunda camada do circuito contém uma única porta neural que calcula:

$$f(x_1, x_2, \ldots, x_n) = sgn\left\{\sum_{k=1}^{s}(y_{k_j} + y'_{k_j}) - s - 1\right\}. \tag{3.12}$$

Para comprovar se a equação fornece a resposta certa, tem-se que para $j = 1,2,3,\ldots s$, se $\sum_{k=1}^{n} x_k \notin \left[q_j, q'_j\right]$, então $y_{k_j} + y'_{k_j} = 1$ para todos os j. Assim:

$$sgn\left\{\sum_{k=1}^{s}(y_{k_j} + y'_{k_j}) - s - 1\right\} = sgn(s - s - 1) = 0. \tag{3.13}$$

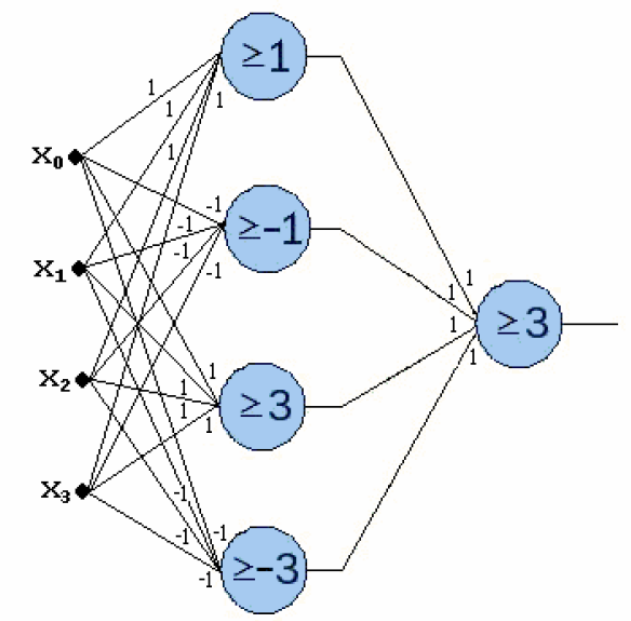
Caso $\sum_{k=1}^{n} x_k \in \left[q_j, q'_j\right]$,então $y_{k_j} + y'_{k_j} = 2$ e $y_{k_i} + y'_{k_i} = 1$ para $i \neq j$. Assim:

$$sgn\left\{\sum_{k=1}^{s}(y_{k_j} + y'_{k_j}) - s - 1\right\} = sgn(s + 1 - s - 1) = 1. \tag{3.14}$$

A primeira camada da rede possui $2s$ portas. Como s é no máximo $\frac{n}{2}$ e a segunda camada possui apenas uma porta, o número total de portas de limiar linear utilizadas é de $n + 1$. Como cda porta no circuito possui *fan-in* de no máximo n, o número de conexões é $O(n^2)$[2].

Por exemplo, na Figura 3.8 temos a implementação da função paridade de quatro variáveis. Esta necessita de 5 portas de limiar linear. Utilizando a lógica tradicional são necessárias 9 portas par a implementação, 8 portas AND e 1 porta OR.

Figura 3.8 - Paridade de quatro variáveis utilizando portas de limiar linear



Utilizando uma técnica chamada de Técnica Telescópica é possível diminuir ainda mais o número de portas de limiar linear.

Lema 3.1 Seja o intervalo $[0, n]$ dividido em $s + 1$ subintervalos $[b_0, b_1 - 1], [b_1, b_2 - 1], \ldots, [b_{s-1}, b_{s-1} - 1], [b_s, n]$, em que $0 = b_0 < b_1 < \cdots < b_k <$ n [14].

Seja $y_i = sgn(\sum_{j=1}^{n} x_j - b_i)$,para todo $i = 1, \ldots, k$. Então:

$$\sum_{j=1}^{k} (a_j - a_{j-1}) y_j = a_m. \tag{3.15}$$

Se $\sum_{j=1}^{n} x_j \in [b_m, b_{m-1} - 1]$, em que $a_0 = 0$ e $a_1, \ldots, a_k$ são números reais arbitrários.

Prova: Como $y_i = sgn(\sum_{j=1}^{n} x_j - b_i) = 1$ se e somente se $\sum_{j=1}^{n} x_j \geq b_i$, portanto, se $\sum_{j=1}^{n} x_j \in [b_m, b_{m+1}]$ então $y_1 = y_2 = \cdots = y_m = 1$ e $y_{m+1} = \cdots = y_k = 0$. Dessa forma, tem-se que:

$$\sum_{j=1}^{k}(a_j - a_{j-1})y_j = \sum_{j=1}^{m}(a_j - a_{j-1}) = a_m. \qquad (3.16)$$

Note que $\sum_{j=1}^{n} x_j \in [b_0, b_1 - 1]$, ou seja $m = 0$, então $y_i = 0$ para todo $i = 1,\ldots,k$ e consequentemente $\sum_{j=1}^{k}(a_j - a_{j-1})y_j = 0 = a_0$.

Utilizando o Lema 3.1, pode-se obter um circuito de limiar de duas camadas com no máximo $\frac{n}{2} + 1$ portas que calculam funções simétricas.

Seja $f(X)$ uma função simétrica de n variáveis. Seja um conjunto de inteiros, s_i e S_i, com $0 \leq s_i \leq S_i \leq n$ para $i = 1,\ldots,\tau$ e $S_i{+}1 < s_{i+1}$ para $i < \tau$, tal que $f(X) = 1$ se e somente se para algum i com:

$$s_i \leq \sum_{j=1}^{n} x_j \leq S_i. \qquad (3.17)$$

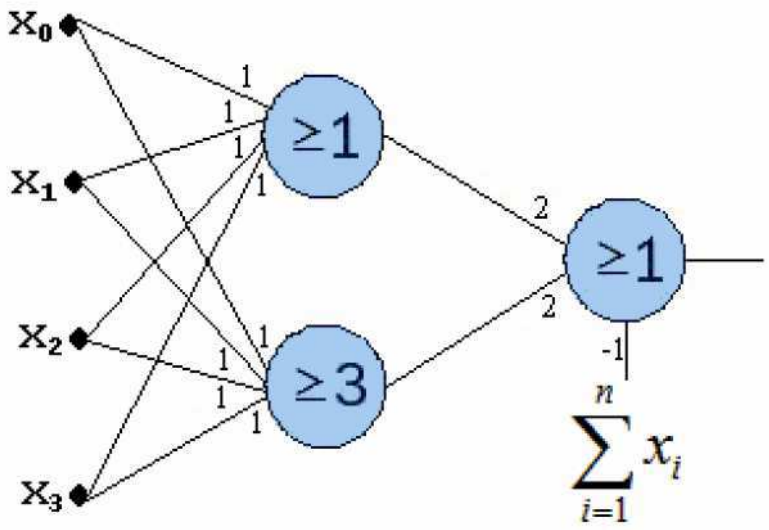
A primeira camada do circuito consiste de τ portas de limiar linear calculando $y_i = sgn(\sum_{j=1}^{n} x_j - s_i)$ para cada i, sendo $1 \leq i \leq \tau$. A segunda camada possui uma única porta que calcula a seguinte função:

$$z = sgn\left\{\sum_{j=1}^{n}(S_j - S_{j-1})\right\}y_j - \sum_{j=1}^{n} x_{j.} \qquad (3.18)$$

Se $f(X) = 1$, então $s_m \leq \sum_{j=1}^{n} x_j \leq S_m$ e, portanto $z = sgn(S_m - \sum_{j=1}^{n} x_j) = 1$. Por outro lado, se $f(X) = 0$, então $s_m \leq \sum_{j=1}^{n} x_j \leq S_{m+1} - 1$ e, portanto, $z = sgn(S_m - \sum_{j=1}^{n} x_j) = 0$.

O número de portas de limiar linear no circuito é $\tau + 1$. Como τ é no máximo $\frac{n}{2}$, então o número máximo de portas de limiar é de $\frac{n}{2} + 1$. Por exemplo, na Figura 3.9 tem-se a implementação da função paridade de quatro variáveis. Aplicada a técnica, esta necessita de 3 portas de limiar linear.

Figura 3.9 - Paridade de quatro variáveis com circuito de duas camadas aplicando a técnica telescópica [7]



3.5.2 Circuitos de três camadas

Generalizando a técnica telescópica para construções de 3 camadas é possível uma redução do número de portas de limiar linear que calculem funções simétricas com $2\sqrt{n} + 1$ portas.

Utilizando um conjunto de inteiros s_i e S_i, em que $i = 1, \ldots, \tau$, com $s_i \leq S_i \leq s_{i+1}$ tal que $f(X) = 1$ se e somente se:

$$s_i \leq \sum_{j=1}^{n} x_j \leq S_i. \tag{3.19}$$

Dividindo o intervalo $[0, n]$ em d subintervalos consecutivos $[s_1, s_{2_1}], [s_2, s_{3_1}], \ldots, [s_d, n]$ tal que cada subintervalo, exceto possivelmente o último, contenha o mesmo número l de inteiros s_i e S_i, em que $l \leq \frac{n}{2d}$. O i-ésimo subintervalo conterá os inteiros $s_{i_1} \leq S_{i_1} \leq s_{i_2} \leq S_{i_2} \leq \cdots S_{i_1} \leq S_{i_1}$. A primeira camada do circuito consiste em d elementos de limiar linear calculando a seguinte função:

$$z_i = sgn(\sum_{j=1}^{n} x_j - s_{i_1}), \tag{3.20}$$

para $i = 1, \dots d$.

Para cada $k = 1, \dots, l$, são definidas duas somas telescópicas:

$$T_k = S_{1_k} z_1 + (S_{2_k} + S_{1_k}) z_2 + (S_{3_k} + S_{2_k}) z_3 + \cdots + (S_{d_k} + S_{d-1_k}) z_d. \tag{3.21}$$

$$t_k = s_{1_k} z_1 + (s_{2_k} + S_{1_k}) z_2 + (s_{3_k} + s_{2_k}) z_3 + \cdots + (s_{d_k} + s_{d-1_k}) z_d. \tag{3.22}$$

Observe que T_k e t_k são combinações lineares das saídas da primeira camada. A segunda camada consiste em 2l portas de limiar linear, cada uma utiliza os inteiros T_k ou t_k como valor de limiar para o cálculo de Q_k e q_k definidos a seguir:

$$Q_k = sgn(T_k - \sum_{j=1}^{n} x_j). \tag{3.23}$$

$$q_k = sgn(t_k - \sum_{j=1}^{n} x_j). \tag{3.24}$$

A terceira camada é única e calcula a seguinte função:

$$f(X) = sgn\{\sum_{k=1}^{l} (Q_k - q_k) - 2l - 1. \tag{3.25}$$

Supondo que $\sum_{j=1}^{n} x_j$ pertence ao m-ésimo intervalo em s e utilizando o Lema 1, a soma telescópica de T_ke t_k assume os seguintes valores:

$$T_k = S_{m_k}. \tag{3.26}$$

$$t_k = s_{m_k}. \tag{3.27}$$

Da Equação (3.19), por definição, $f(X) = 1$ se, e somente se, para algum *k* obtém-se:

$$s_{m_k} \leq \sum_{j=1}^{n} x_j \leq S_{m_k}. \tag{3.28}$$

Na segunda camada $\sum_{j=1}^{n} x_j$ é comparado com $T_k = S_{m_k}$e $t_k = s_{m_k}$, para cada k. Como $s_{m_i} \leq \sum_{j=1}^{n} x_j \leq S_{m_i}$, o valor da saída da segunda camada (Q_k, q_k) pode ser visto como sendo:

$$Q_k + q_k = f(x) = \begin{cases} 2, & se\ k = i \\ 1, & se\ k \neq i. \end{cases} \tag{3.29}$$

Logo, o elemento de saída da terceira camada é:

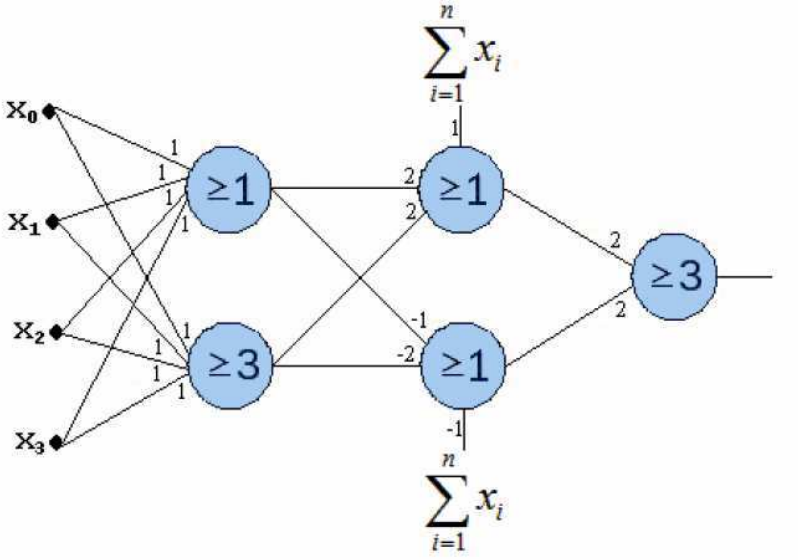
$$sgn\{\sum_{k=1}^{l} 2(Q_k + q_k) - 2l - 1 = sgn(2l + 2 - 2l - 1) = 1. \tag{3.30}$$

Se $f(X) = 0$, então não existe um k tal que a inequação (3.28) seja satisfeita. Portanto, $Q_k + q_k = 1$ para todo k e a porta da terceira camada calcula a seguinte expressão:

$$sgn\{\sum_{k=1}^{l} 2(Q_k + q_k) - 2l - 1 = sgn(2l - 2l - 1) = 0. \tag{3.31}$$

O circuito fornece a saída correta para qualquer entrada $X = [x_1, x_2, \ldots, x_n]$. A primeira camada do circuito consiste em d elementos. A segunda camada possui $2l \leq \frac{n}{d} + 1$ portas e a terceira apenas uma porta. Como $d = \sqrt{n}$, então o tamanho do circuito de 3 camadas é de $2\sqrt{n}$+1 portas de limiar linear. Na Figura 3.10 é mostrado um diagrama de implementação de um circuito que computa a paridade de 4 variáveis em um circuito de 3 camadas. Esta configuração possui 5 portas de limiar linear. Para um número maior de variáveis, a implementação utilizando circuitos de 3 camadas tende a possuir menos portas de limiar linear quando comparado a implementação utilizando circuitos de 2 camadas, embora o tempo de atraso aumente.

Figura 3.10 - Implementação de um circuito com 3 camadas para computar a paridade [7]



O crescimento do tamanho circuito é $2^{n-1}+1$ utilizando a lógica tradicional. Nota-se que o crescimento do número de portas com o tamanho do corpo é exponencial. Neste capítulo foram apresentadas técnicas para a construção de circuitos de limiar de duas que computam qualquer função booleana simétrica com, no máximo, $n+1$ e $\frac{n}{2}+1$ e de três camadas que utilizam $2\sqrt{n}+1$ portas. No Quadro 3.2 tem-se o comparativo do crescimento numérico do número de portas para alguns valores de n em projetos de circuitos com uma, duas e três camadas. O crescimento com apenas uma camada corresponde ao pior caso da implementação utilizando a lógica neural, pois as portas tradicionais são subconjuntos das portas neurais. Utilizando as portas neurais, aumentando o número de camadas e fazendo o uso da técnica telescópica, temos que o número de portas utilizadas para computar funções é cada vez menor, quando comparada com as outras implementações.

Quadro 3.2 - Crescimento do número de portas para diferentes tamanhos de corpo

Entrada	**1 Camada**	**2 Camadas**	**2 Camadas/Telescópica**	**3 Camadas/Telescópica**
	$2^{n-1}+1$	$\boldsymbol{n+1}$	$\frac{n}{2}+1$	$2\sqrt{n}+1$
0	1	1	1	1
1	2	2	2	3
2	4	3	2	4
3	8	4	3	5
4	16	5	3	5
5	32	6	4	6
7	128	8	5	7
8	**256**	**9**	**5**	**7**
9	512	10	6	7
10	1024	11	6	8
11	2048	12	7	8
12	4096	13	7	8
13	8192	14	8	9
14	16384	15	8	9
15	32768	16	9	9
16	65536	17	9	9

3.6 Conclusão

Neste capítulo foi apresentada a rede neural utilizada para realizar a multiplicação em corpos finitos e a estrutura básica da mesma, as portas de limiar linear. Estas possuem a vantagem de terem as portas tradicionais (AND, OR e NOT) como um subconjunto. Verifica-se a diminuição da complexidade espacial de funções simétricas de um crescimento exponencial para um crescimento polinomial. Com o uso de técnicas algébricas, como a técnica telescópica, podemos extrair esta vantagem da porta de limiar linear e construir circuitos de limiar linear com 2 camadas utilizando $n+1$ e $\frac{n}{2}+1$ portas e de 3 camadas utilizando $2\sqrt{n}+1$ portas, em substituição ao circuito tradicional com $2^{n-1}+1$ portas. Esta característica pode ser aproveitada para o projeto e implementação de circuitos utilizando redes neurais que realizem operações aritméticas em corpos finitos mais eficientes, já que esta característica é amplamente explorada na computação de adição, multiplicação e divisão em aritmética tradicional e aplicações como criptografia e códigos corretores de erros. [27,28].

3. MULTIPLICADOR EM GF(2^n) UTILIZANDO PORTAS DE LIMIAR LINEAR

4. Multiplicador em $GF(2^n)$ Utilizando Portas de Limiar Linear

Neste capítulo apresenta-se uma nova arquitetura para multiplicadores em $GF(2^n)$ utilizando portas de limiar linear. Esta arquitetura é mais eficiente do que a apresentada na Seção 2.2.

A principal ideia é realizar a operação da multiplicação polinomial e a redução de módulo em um único passo. Devido às características da porta de limiar linear, esta arquitetura necessita de menos portas e torna o atraso constante para a operação com qualquer corpo.

Ao final das seções, mostra-se o circuito de limiar linear correspondente à multiplicação em $GF(2^4)$ e $GF(2^8)$.

4.1 Introdução

Redes neurais artificiais de multicamadas podem ser usadas para construção do multiplicador $GF(2^n)$. No trabalho de *Lidiano* [1], foi projetada teoricamente a rede, mostrando não apenas a viabilidade do ponto de vista tecnológico, como também melhorias em termos de consumo de *hardware* à medida que o número de entradas for crescendo.

Algumas funções booleanas, tais como adição e multiplicação, só podem ser calculadas utilizando as portas AON com número de camadas fixas se o tamanho aumentar exponencialmente. Isso resulta em uma área de *chip* que também aumenta exponencialmente [7]. Para que o tamanho do *chip* aumente polinomialmente é necessário projetar circuitos com retardo não fixo, o que acarreta em circuitos mais lentos. Portanto, são ineficientes para algumas aplicações. Com o uso de redes neurais é possível obter circuitos com profundidade fixa e tamanho polinomial.

Como algumas aplicações a velocidade é a prioridade, a utilização de circuitos cuja profundidade crescente em função do número de entradas não é desejada. Portanto, é de grande valia o compromisso entre o tamanho e a profundidade do circuito, de modo que sejam obtidas operações mais eficientes para determinada aplicação. Utilizando a lógica neural é possível se obter arquiteturas, para a multiplicação, que tenha um crescimento polinomial. Desta forma, é possível ter um baixo número de portas e um circuito mais rápido.

4.2 Multiplicador de Mastrovito Utilizando Porta de Limiar Linear

Para o projeto deste multiplicador utiliza-se o multiplicador de Mastrovito, descrito na Seção 2.2, como base. A diferença entre este e o apresentado é que ao invés de utilizar portas tradicionais serão utilizadas portas de limiar linear.

4.2.1 Multiplicação Polinomial Ordinária

Nesta subseção é utilizado o mesmo procedimento utilizado para a multiplicação em $GF(2^n)$ na Seção 2.2. Para a primeira camada do multiplicador são necessárias as mesmas n^2 portas de limiar linear que seriam utilizadas na lógica tradicional, cada porta realizando a operação AND *bit* a *bit* para o cálculo da multiplicação de cada entrada $a_i b_k$. Contudo, para a segunda camada, o quantitativo necessário para realizar operações XOR pode ser dado utilizando a técnica telescópica [2].

Na subseção 3.5.1, tem-se que circuitos de 2 camadas e n variáveis necessitam de $\frac{n}{2}+1$ portas de limiar com *fan-in* ilimitado. Seja $\#TG$ o número de portas de limiar linear para implementar a multiplicação. Para $n > 2$, temos:

$$\#TG = 2.\sum_{i=2}^{n}\left[\left(\frac{i}{2}+1\right)-\left(\frac{n}{2}+1\right)\right]. \tag{4.1}$$

Caso n seja par, o número de portas é dado por:

$$\#TG = 2.\sum_{i=2}^{n}\left[\left(\frac{i}{2}+1\right)+2(n-1)-\frac{n}{2}-1\right]. \tag{4.2}$$

$$\#TG' = 2.\left(\frac{n^2}{2}+\frac{n}{2}-1\right)+2(n-1)-\frac{n}{2}-1. \tag{4.3}$$

$$\#TG' = \left(\frac{n^2}{2}+\frac{5n}{2}-5\right). \tag{4.4}$$

Caso n seja ímpar, o número de portas é dado por:

$$\#TG' = 2.\sum_{i=2}^{n}\left[\left(\frac{i}{2}\right)+2(n-1)-\frac{n}{2}-\frac{1}{2}-1\right]. \tag{4.5}$$

$$\#TG' = 2.\left(\frac{n^2}{4}+\frac{n}{2}-\frac{3}{4}\right)+\frac{3n}{2}-\frac{7}{2}. \tag{4.6}$$

$$\#TG' = \left(\frac{n^2}{2} + \frac{5n}{2} - 5\right). \tag{4.7}$$

Logo, para qualquer n, tem-se:

$$\#TG' = \left(\frac{n^2}{2} + \frac{5n}{2} - 5\right). \tag{4.8}$$

O atraso da operação é de $3\tau_{TG}$. O retardo $2\tau_{TG}$ é gasto para realizar as somas XOR e $1\tau_{TG}$ é necessário para implementar a porta AND.

4.2.2 Redução de Módulo $p(x)$

Para realizar a redução de módulo $p(x)$ é utilizada a Equação (2.34). Logo, o número de portas de limiar linear é dado por:

$$\#TG'' = 2 + \sum_{j=1}^{n-2} 3 + 2. \tag{4.9}$$

$$\#TG'' = 2 + 3(n-2) + 2. \tag{4.10}$$

$$\#TG'' = 3n - 2. \tag{4.11}$$

Portanto, o número total de portas é:

$$\#TG = \#TG' + \#TG'' = (3n-2) + \left(\frac{n^2}{2} + \frac{5n}{2} - 5\right). \tag{4.12}$$

$$\#TG = \left(\frac{3n^2}{2} + \frac{11n}{2} - 7\right). \tag{4.13}$$

São necessárias 2 camadas para o cálculo da redução de módulo $p(x)$. Logo o atraso é de $2\tau_{TG}$ para este segundo passo e de $5\tau_{TG}$ o atraso total do multiplicador.

4.3 Nova Arquitetura de Multiplicador

Nesta seção é apresentada uma nova arquitetura mais eficiente do que as apresentadas nas seções 2.2 e 4.2. Utilizando a técnica telescópica e devido às características das portas de limiar linear é possível reduzir ainda mais a quantidade de portas de limiar utilizadas e retardo total do multiplicador. A principal ideia está na realização da multiplicação polinomial e redução de módulo em um único passo. Considere que pode-se reescrever a equação (2.27) como segue:

$$C(x) = C'(x) mod\ P(x). \tag{4.14}$$

$$C(x) = (c'_0 + c'_n) + \sum_{j=1}^{n-2}\left(c'_j + c'_{j+n-1} + c'_{j+n}\right)x^j + (c'_{n-1} + c'_{2n-2})x^{n-1}. \quad (4.15)$$

Substituindo cada c'_j pelos seus respectivos valores, dados na Equação (2.31), resulta na seguinte equação:

$$\begin{aligned} C(x) = &\left(a_0 b_0 + \sum_{i=0}^{n-1} a_i b_{n-1}\right) \\ &+ \sum_{j=1}^{n-2}\left(\sum_{i=1}^{j} a_i b_{j-i} + \sum_{i=j}^{n-1} a_i b_{j+n-1-i} + \sum_{i=j+1}^{n-1} a_i b_{j+n-i}\right)x^j \\ &+ \left[\left(\sum_{i=0}^{n-1} a_i b_{n-1-i}\right)x^{n-1} + a_{n-1}b_{n-1}\right]. \end{aligned} \quad (4.16)$$

O número de portas de limiar linear para o cálculo de cada coeficiente da equação (4.16) é dado por:

$$\#c_0 = 1 + n - 1 = n$$

$$\begin{aligned} \#c_j &= (j+1) + 2n - j - n + 1 - 1 + 2n - j - n - 1 \\ \#c_j &= 2n - j, \#c_{n-1} = n + 2n - 2n + 2 = n + 1, \end{aligned} \quad (4.17)$$

em que $j = 1,2,\ldots,n-2$.

Assim, o número total de portas de limiar linear necessárias para a multiplicação em corpos finitos é dado por:

$$\#TG = \left(\frac{n}{2}\right) + 1 + \sum_{j=1}^{n-2}\left[\left(\frac{2n-j}{2}\right) + 1\right] + \left(\frac{n+1}{2}\right) + 1. \quad (4.18)$$

$$\#TG = \left(\frac{n+1}{2}\right) + \sum_{j=1}^{n-2}\left[\left(\frac{2n-j}{2}\right) + 1\right] + \left(\frac{n+1}{2} + 1\right). \quad (4.19)$$

$$\#TG = n + 3 + \sum_{j=1}^{n-2}\left[\left(\frac{2n-j}{2}\right) + 1\right] + 1. \quad (4.20)$$

$$\#TG = n + 3 + (n-2)\sum_{j=1}^{n-2}\left[\left(\frac{2n-j}{2}\right)\right]. \quad (4.21)$$

$$\#TG = 2n + 1 + \sum_{j=1}^{n-2} \left[\left(\frac{2n - j}{2} \right) \right]. \tag{4.22}$$

Para n par, tem-se:

$$\#TG = 2n + 1 + \left(\frac{3n^2}{4} - n + 1 \right). \tag{4.23}$$

$$\#TG = \left(\frac{3n^2}{4} + n \right). \tag{4.24}$$

Para n ímpar, tem-se:

$$\#TG = 2n + 1 + \left(\frac{3n^2}{4} - n + \frac{3}{4} \right). \tag{4.25}$$

$$\#TG = \left(\frac{3n^2}{4} + n + \frac{1}{4} \right). \tag{4.26}$$

Somando os resultados das equações (4.24) e (4.26) com o número de portas da primeira camada, que é n^2, tem-se o total de portas necessárias para a multiplicação em $GF(2^n)$:

$$\#TG = \begin{cases} \frac{7}{4}n^2 + \ n\,, & n\ para\ n\ par \\ \frac{7}{4}n^2 + n + \frac{1}{4}\,, & para\ n\ impar \end{cases} \tag{4.27}$$

O atraso do Novo Multiplicador é $3\tau_{TG}$, não importando o tamanho do corpo.

No Quadro 4.1 e na Figura 4.1 tem-se a evolução do crescimento do número de portas em relação ao tamanho n do corpo.

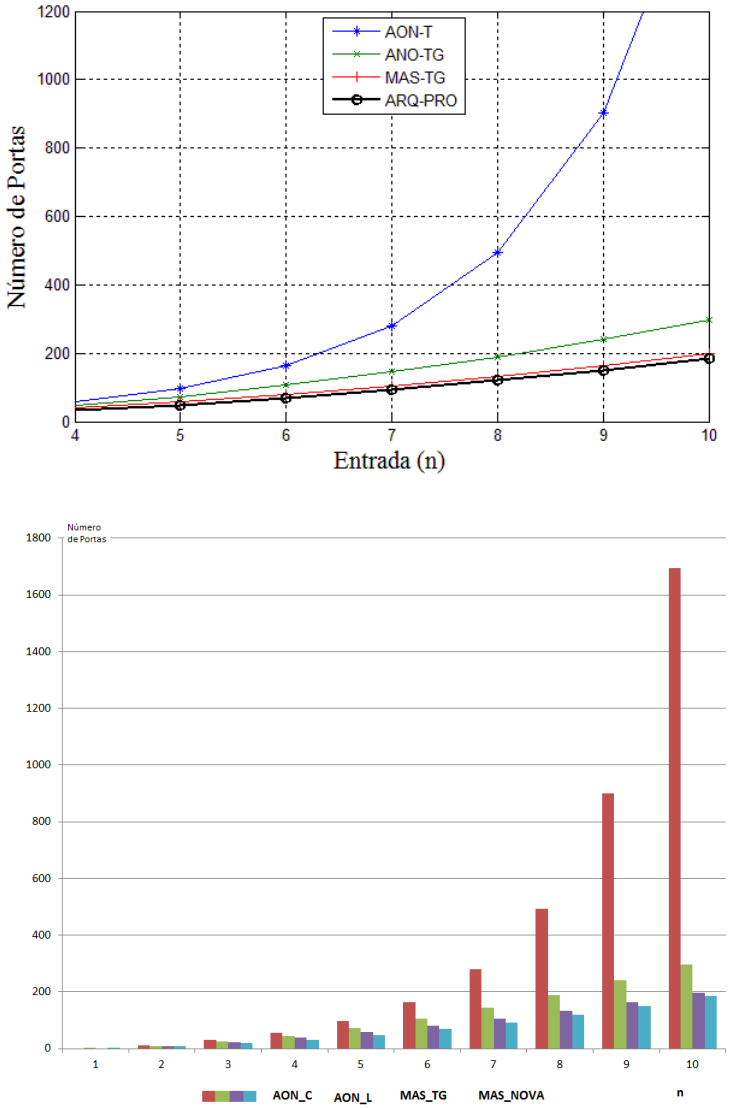
Quadro 4.1 - Evolução do número de portas com o tamanho do corpo

GF(2^n)	**AON-T**	**AON-TG**	**MAS-TG**	**ARQ-PRO**
n	$3.2^{n-1} + n^2 + 7n - 11$	$3n^2 - 2$	$\frac{3}{2}n^2 + \frac{11}{2}n - 7$	$\frac{7}{4}n^2 + n + \frac{1}{4}$
1	0	1	0	3
2	13	10	10	9
3	31	25	23	19
4	57	46	39	32
5	97	73	58	49
6	163	106	80	69
7	279	145	105	93
8	493	190	133	120
9	901	241	164	151
10	1695	298	198	185
11	3259	361	235	223
12	6361	430	275	264
13	12541	505	318	309
14	24859	586	364	357
15	49475	673	413	409
16	98661	766	465	464

Em que:

- AON-T: Arquitetura Tradicional;
- AON-TG: Arquitetura Tradicional com Portas de Limiar;
- MAS-TG: Arquitetura de Mastrovito com Portas de Limiar;
- ARQ-PRO: Nova Arquitetura proposta com Portas de Limiar Linear;
- n: Tamanho do corpo.

Figura 4.1 - Comparativo do crescimento exponencial com o polinomial



Na Figura 4.1, a curva construída em azul representa uma implementação utilizando as portas tradicionais AON. A curva plotada em verde representa a implementação do limiar linear em portas tradicionais AON. A curva plotada em vermelho (MAS-TG), é uma implementação usando portas de limiar conforme Mastrovito [19]. Já a curva em preto, é para uma implementação otimizada de MAS-TG, conforme em [26]. Pode-se perceber o crescimento exponencial para as portas AON e um comportamento polinomial para implementações que fazem uso do limiar linear e da rede neural discreta. Logo, conclui-se que à medida que o número n aumenta, as diferenças em termos de portas se tornam extremamente elevadas. Logo, para $n = 8$, temos uma diferença no que se refere ao número de portas utilizadas na

implementação do multiplicador utilizando portas tradicionais e utilizando a arquitetura proposta.

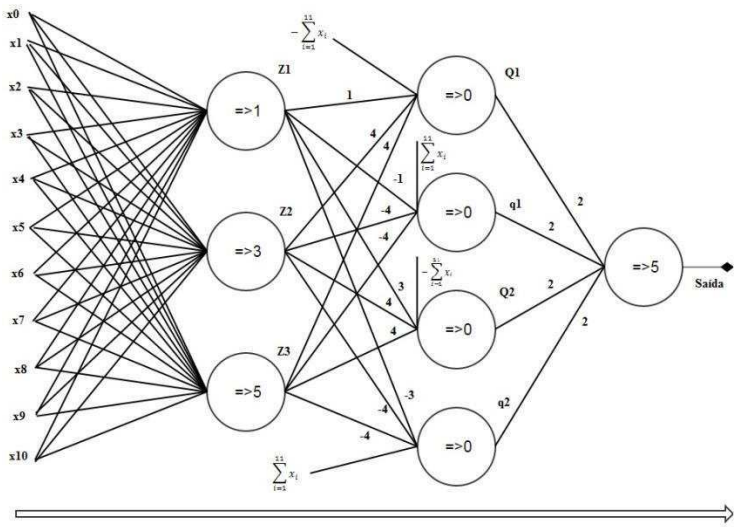
É possível, como mostrado em [7], com a utilização de redes neurais discretas e ao tornar a profundidade ilimitada, reduzir o tamanho dos circuitos de exponencial para polinomial. Assim, é possível a utilização de arquiteturas de multiplicadores em corpos finitos projetados com portas de limiar linear cujo tamanho não cresça exponencialmente com o número de entradas.

Na Figura 4.2 ilustra-se o procedimento para a realização da função paridade de 11 variáveis. A saída da função paridade é 1 caso $\sum_{i=1}^{n} x_i$ seja ímpar. Fazendo $s_1 = S_1 = 1,$ $s_2 = S_2 = 3, s_3 = S_3 = 5, s_4 = S_4 = 7, s_5 = S_5 = 9\ e\ s_6 = S_6 = 11.$ Utilizando a definição de construção de circuitos de 3 camadas apresentada na seção 3.5.2 e as equações da mesma seção, temos $s_{1_1} = 1$, $s_{2_1} = 5$ e $s_{3_1} = 9$, escolhendo $d = 3$. Logo, temos, em outras palavras:

$$\begin{aligned} s_{1_1} = S_{1_1} = 1 < s_{1_2} = S_{1_2} = 3; \\ s_{2_1} = S_{2_1} = 5 < s_{2_2} = S_{2_2} = 7; \\ s_{3_1} = S_{3_1} = 9 < s_{3_2} = S_{3_2} = 11; \end{aligned} \tag{4.28}$$

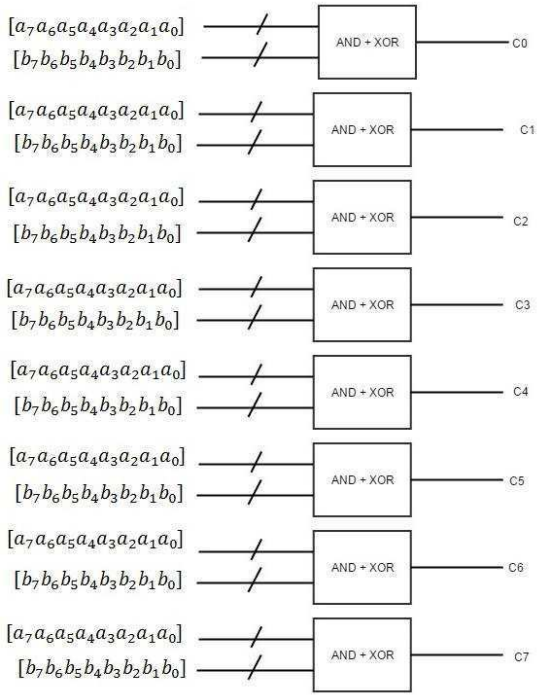
Baseado nessas escolhas, a realização da função paridade de 11 variáveis é pode ser visualizada na Figura 4.2. Esta configuração de 3 camadas exige o portas de limiar linear. Uma implementação com 2 camadas exige 7 portas de limiar linear.

Figura 4.2 - Função Paridade com de 11 variáveis em um circuito com 3 camadas.



Entretanto, com o crescimento do número de variáveis a eficiência do projeto com 3 camadas se torna evidente. Por exemplo, para a realização da função paridade de 36 variáveis em um circuito com 2 camadas requer 10 portas. Já na configuração em circuitos de 3 camadas, requer 13 portas. A implementação destes circuitos e da função paridade é realizada a partir de portas AND e XOR, multiplicação e soma em um corpo finito, como ilustrada no diagrama de blocos da Figura 4.3 para $GF(2^8)$.

Figura 4.3 - Diagrama de Blocos da implementação a partir das operações AND e XOR



O multiplicador $GF(2^8)$ é projetado a partir do desenvolvimento das equações (4.16) e (4.29), para $n = 8$ e $1 \leq j \leq 6$.

$$C(x) = \left(a_0b_0 + \sum_{i=0}^{7} a_i b_{n-1}\right) + \sum_{j=1}^{6}\left(\sum_{i=1}^{j} a_i b_{j-i} + \sum_{i=j}^{7} a_i b_{j+n-1-i} + \sum_{i=j+1}^{7} a_i b_{j+n-i}\right)x^j + \left[\left(\sum_{i=0}^{7} a_i b_{n-1-i}\right)x^7 + a_{n-1}b_{n-1}\right]. \quad (4.29)$$

Resolvendo todos os termos, temos a saída $C(x)$ representada na Equação (4.30) como uma combinação das entradas $a_i b_k$:

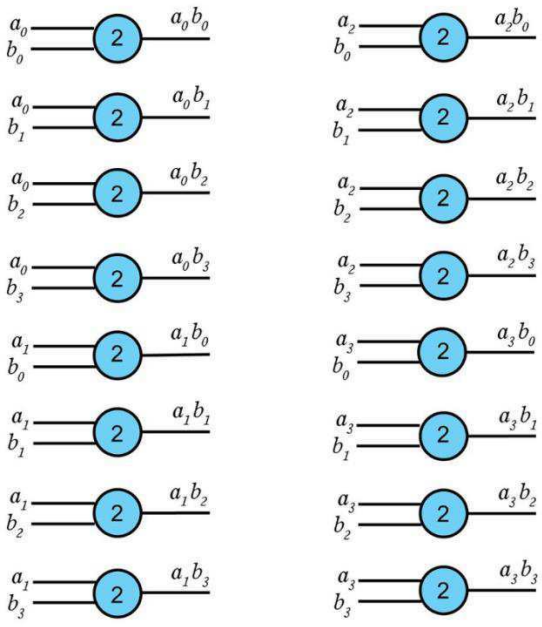
$$
\begin{aligned}
C(x) = \; & (a_0b_0 + a_0b_7 + a_1b_7 + a_2b_7 + a_3b_7 + a_4b_7 + a_5b_7 + a_6b_7 + a_7b_7)x^0 + \\
& + (a_1b_0 + a_1b_7 + a_2b_6 + a_3b_5 + a_4b_4 + a_5b_3 + a_6b_2 + a_7b_1)x^1 + \\
& + (a_1b_1 + a_2b_0 + a_2b_7 + a_3b_6 + a_4b_5 + a_5b_4 + a_6b_3 + a_7b_2 + a_3b_7 + a_4b_6 + a_5b_5 + a_6b_4 + a_7b_3)x^2 + \\
& + (a_1b_2 + a_2b_1 + a_3b_0 + a_3b_7 + a_4b_6 + a_5b_5 + a_6b_4 + a_7b_3 + a_4b_7 + a_5b_6 + a_6b_5 + a_7b_4)x^3 + \\
& + (a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + a_4b_7 + a_5b_6 + a_6b_5 + a_7b_4 + a_5b_7 + a_6b_6 + a_7b_5)x^4 + \\
& + (a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0 + a_5b_7 + a_6b_6 + a_7b_5 + a_6b_7 + a_7b_6)x^5 + \\
& + (a_1b_5 + a_2b_4 + a_3b_3 + a_4b_2 + a_5b_1 + a_6b_0 + a_6b_7 + a_7b_6 + a_7b_7)x^6 + \\
& + (a_0b_7 + a_1b_6 + a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2 + a_6b_1 + a_7b_0 + a_7b_7)x^7 \quad .
\end{aligned}
\tag{4.30}
$$

Repetindo o desenvolvimento das equações (4.16) e (4.29), tem-se, para $n = 4$ e $1 \leq j \leq 2$, tem-se a saída $C(x)$ para o multiplicador em $GF(2^4)$:

$$
\begin{aligned}
C(x) = \; & (a_0b_0 + a_1b_3 + a_2b_2 + a_3b_1)x^0 + \\
& + (a_1b_3 + a_2b_2 + a_2b_3 + a_3b_1 + a_3b_2)x^1 + \\
& + (a_0b_2 + a_1b_1 + a_2b_0 + a_2b_3 + a_3b_2 + a_3b_3)x^2 + \\
& + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + a_3b_3)x^3 .
\end{aligned}
\tag{4.31}
$$

Eq.(4.30) pode ser implementada utilizando circuitos analógicos e circuitos digitais, utilizando as mais diversas técnicas. Uma delas é realizando a função paridade em uma rede neural discreta, definida como 0 se o somatório das suas entradas for par e 1 caso contrário.

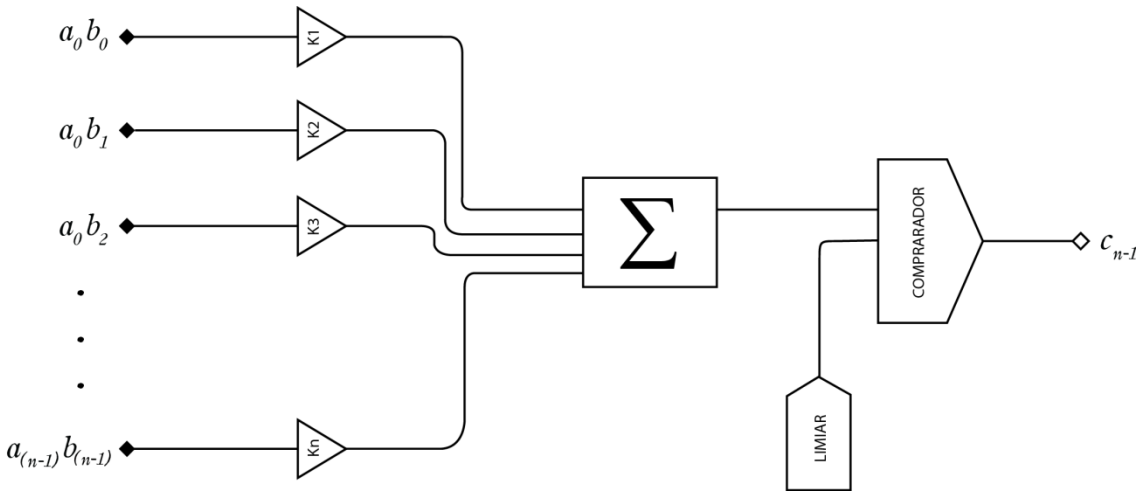
Figura 4.4 - Primeira camada para o multiplicador em $GF(2^4)$



Para $GF(2^4)$ são necessárias 16 portas de limiar que calculam um AND bit a bit para o cálculo de cada a_ib_k da primeira camada. São necessárias 16 e 64 portas lógicas para a realização da primeira camada do multiplicador em $GF(2^4)$ e $GF(2^8)$,

respectivamente. A segunda e terceira camada é um circuito que computa a paridade com relação aos coeficientes $a_i b_k$ de cada saída. São necessárias 16 e 51 portas de limiar para o cálculo de todos os coeficientes da expressão de saída. Cada paridade é calculada utilizando portas de limiar linear em um circuito de limiar, como se pode verificar na Figura 4.5.

Figura 4.5 - Diagrama da Porta de Limiar Linear para o multiplicador em $GF(2^n)$ utilizando redes neurais discretas



Para o cálculo da multiplicação de todos os $a_i b_j$ é utilizado o mesmo conjunto de portas da figura 4.4. Nas figuras 4.6, 4.7 e 4.8 são mostrados os diagramas da implementação de cada bit c_i de $C(x)$.

Figura 4.6 - Diagrama de implementação do bit c_0 para $GF(2^4)$

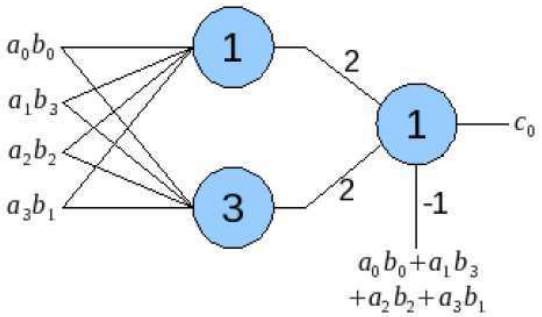


Figura 4.7 - Diagrama de implementação do bit c_1 para $GF(2^4)$

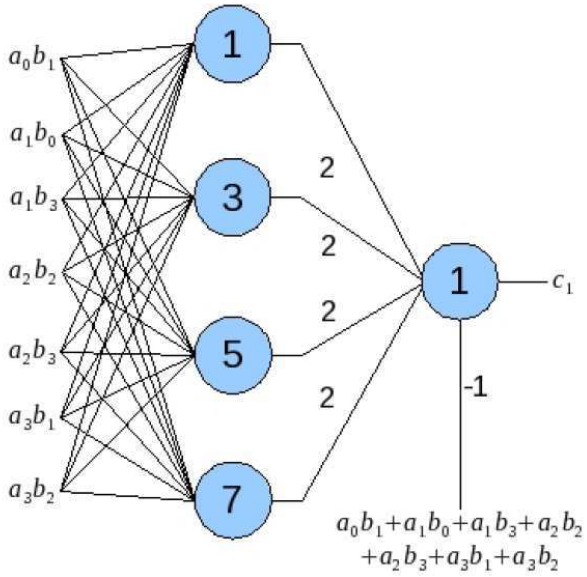


Figura 4.8 - Diagrama de Implementação do bit c_2 para $GF(2^4)$

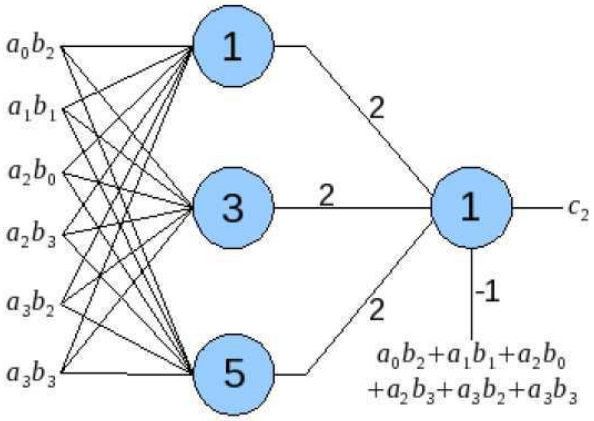
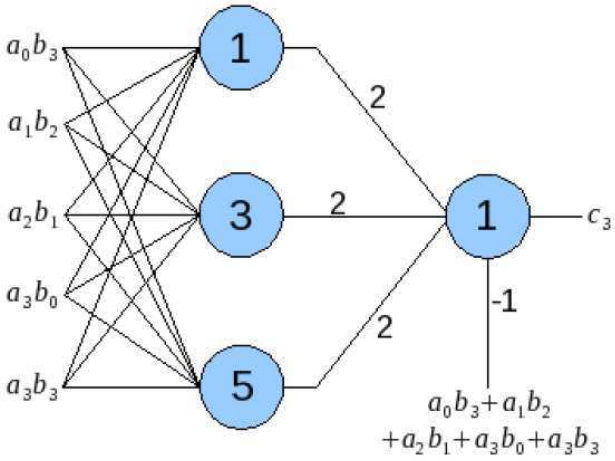
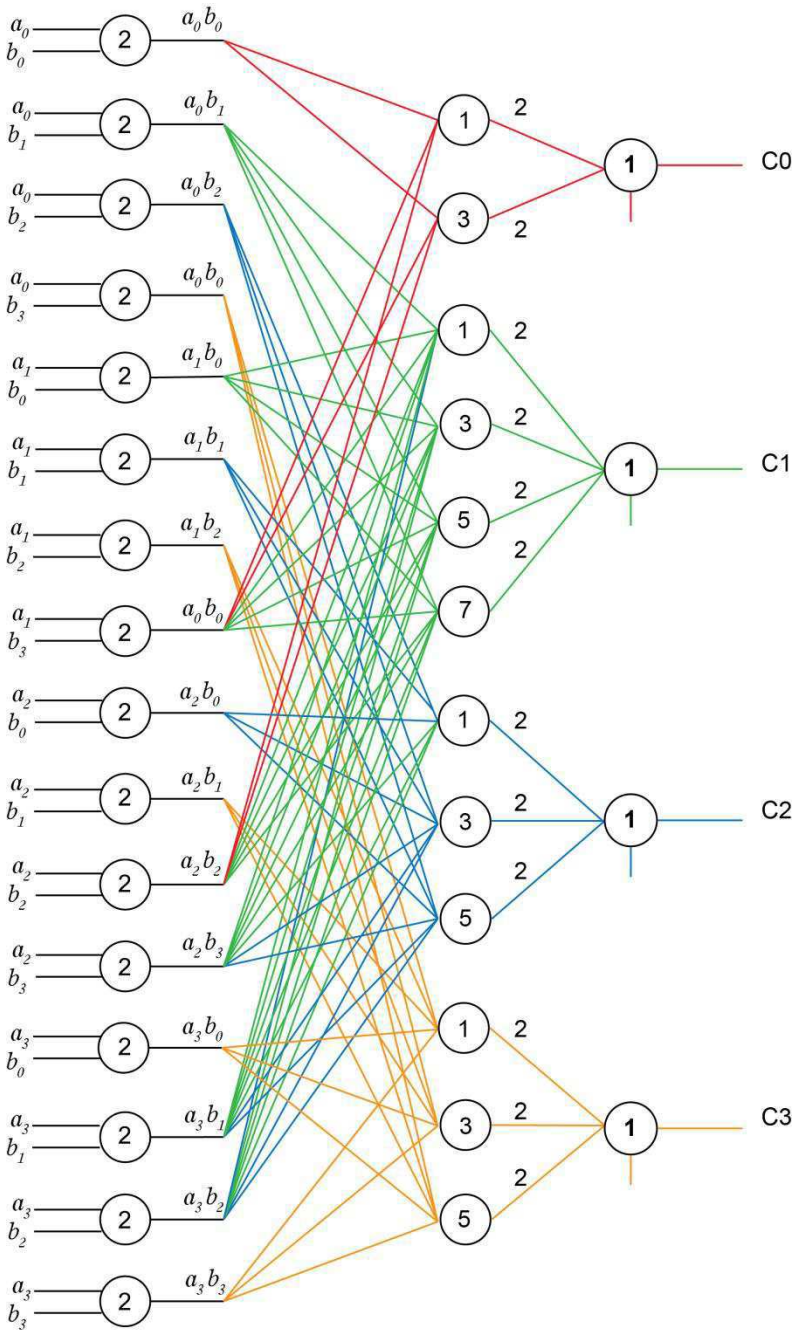


Figura 4.9 - Diagrama da Implementação do bit c_3 para $GF(2^4)$



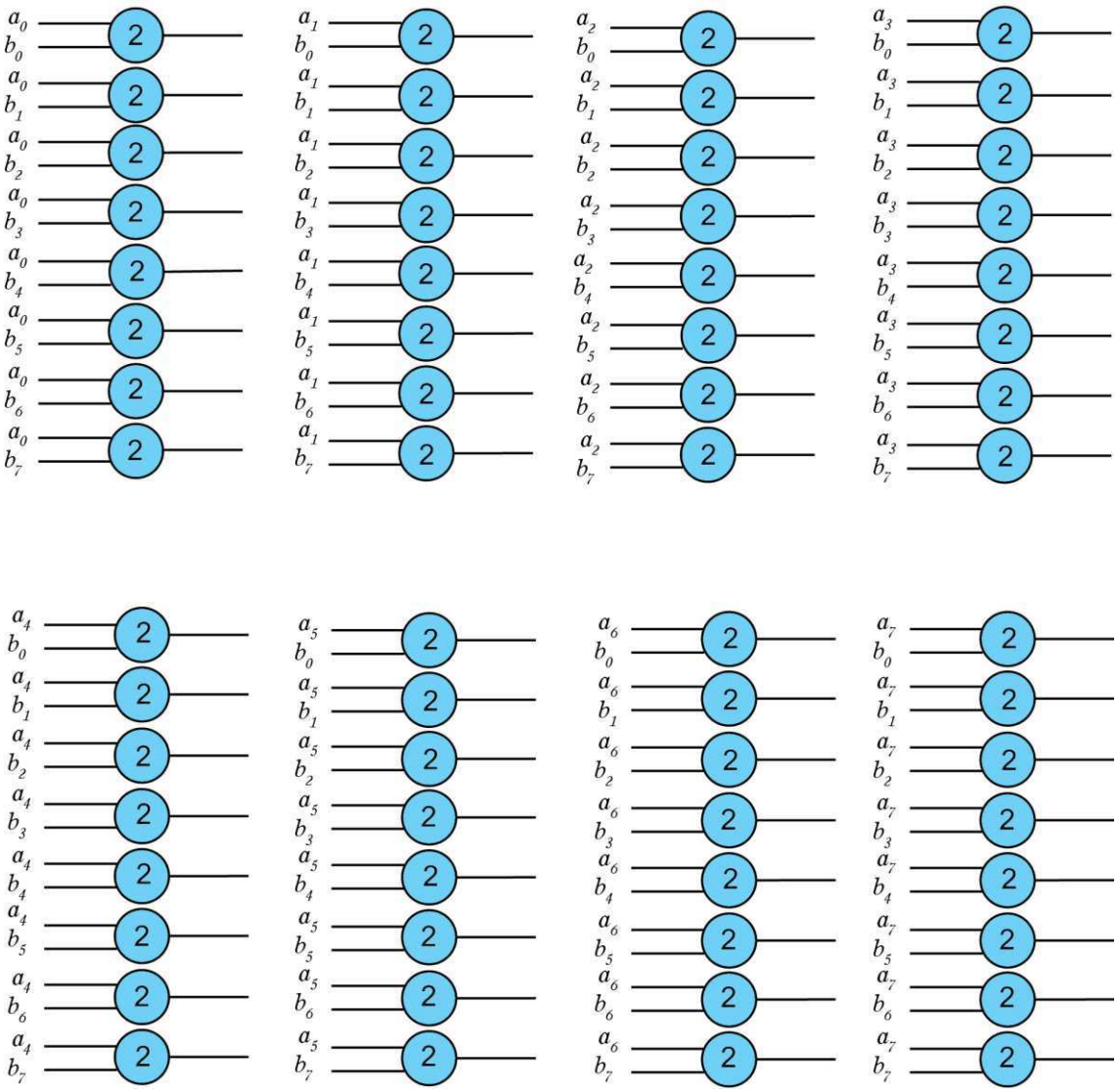
Como pode ser observado na figura 4. 10, o multiplicador em $GF(2^4)$ possui 3 camadas. A primeira camada realiza a multiplicação bit a bit $a_i b_j$, para $i = 0,\ldots,3$ e $j = 0,\ldots,3$. A segunda e terceira camadas é um circuito que calcula a paridade em relação aos coeficientes $a_i b_j$ de cada c_i. O diagrama geral do multiplicador em $GF(2^4)$ utilizando portas de limiar pode ser visualizado na Figura 4.10.

Figura 4.10 - Multiplicador em $GF(2^4)$ utilizando portas de limiar linear



A primeira camada do multiplicador em $GF(2^8)$ é computada por 64 portas lógicas que computam um AND *bit* a *bit*, como pode ser visto na Figura 4.11.

Figura 4.11 - Primeira camada para o multiplicador em $GF(2^8)$



Nas figuras 4.12, 4.13 e 4.14, 4.15, 4.16, 4.17, 4,18 e 4.19 são mostrados os diagramas da implementação de cada bit c_i de $C(x)$, em que computam a função paridade de 8, 10, 10, 11, 12 e 13 variáveis de entrada.

Figura 4.12 - Diagrama da implementação do bit c_0 para $GF(2^8)$

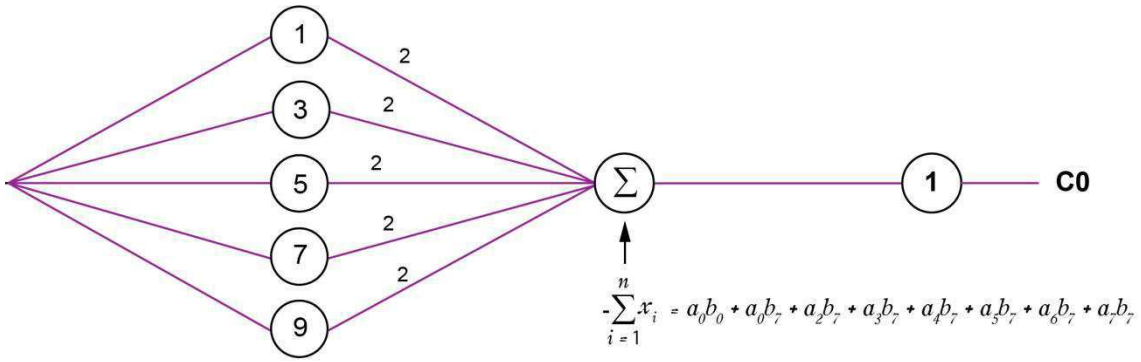


Figura 4.13 - Diagrama da implementação do bit c_1 para $GF(2^8)$

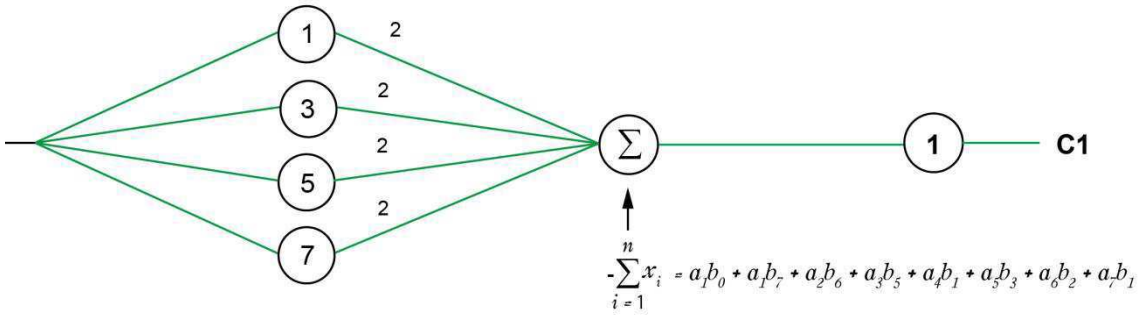


Figura 4.14 - Diagrama da implementação do bit c_2 para $GF(2^8)$

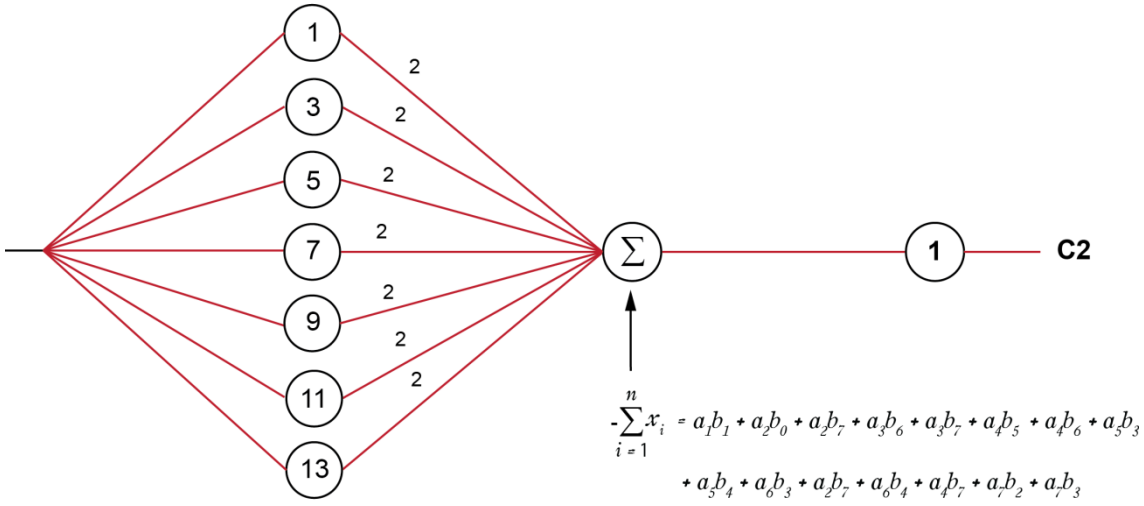


Figura 4.15 - Diagrama da implementação do bit c_3 para $GF(2^8)$

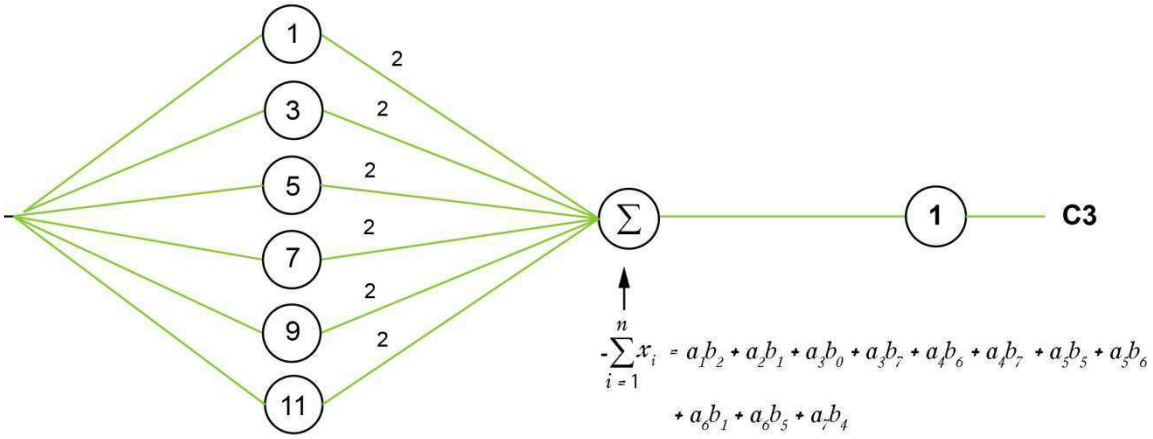


Figura 4.16 - Diagrama da implementação do bit c_4 para $GF(2^8)$

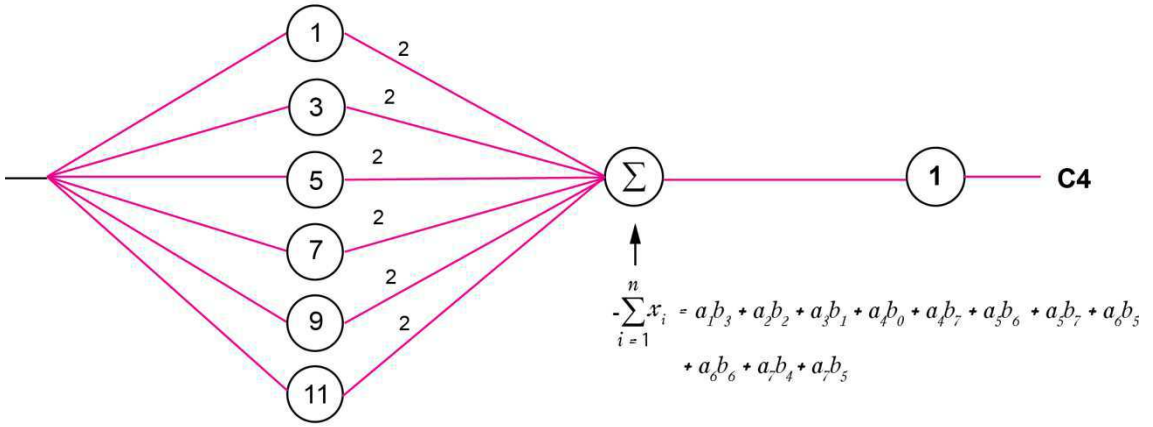


Figura 4.17 - Diagrama da implementação do bit c_5 para $GF(2^8)$

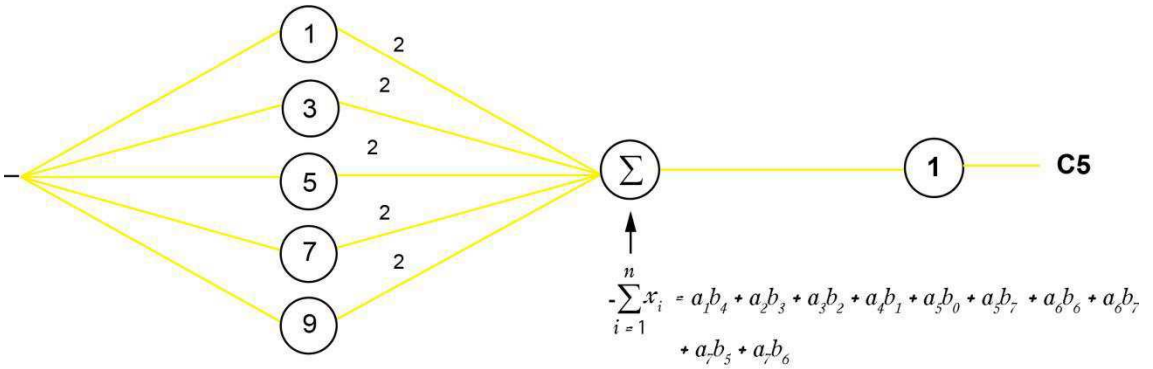


Figura 4.18 - Diagrama da implementação do bit c_6 para $GF(2^8)$

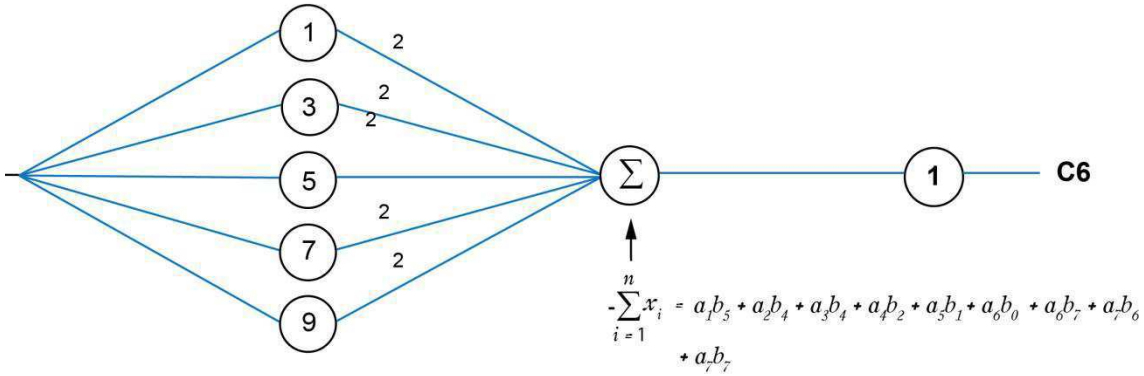
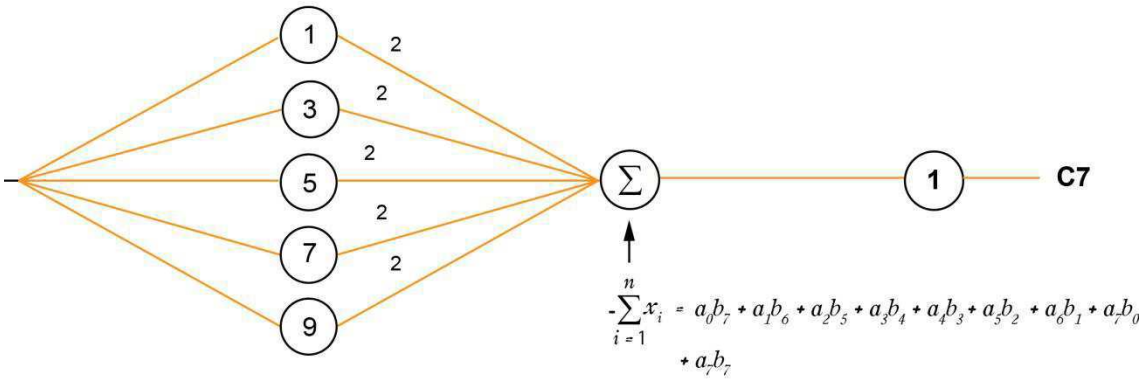
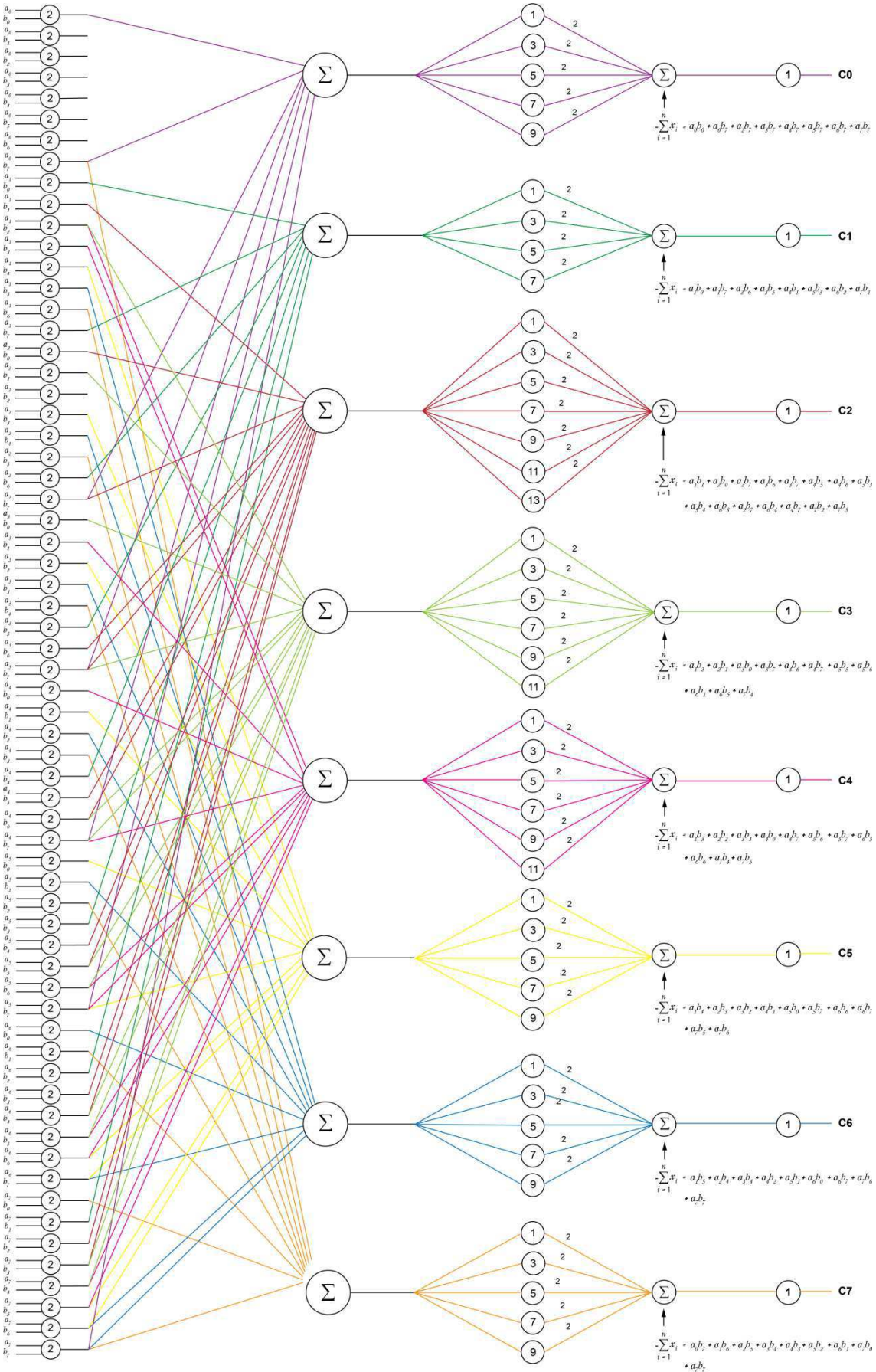


Figura 4.19 - Diagrama da implementação do bit c_7 para $GF(2^8)$



Como pode ser observado na figura 4. 20, o multiplicador em $GF(2^8)$ possui 3 camadas. A primeira camada realiza a multiplicação bit a bit a_ib_j, para $i = 0,\ldots,7$ e $j = 0,\ldots,7$. A segunda e terceira camadas é um circuito que calcula a paridade em relação aos coeficientes a_ib_j de cada c_i. Os pesos das conexões e os limiares das portas foram previamente calculadores utilizando a técnica descrita na seção 5.5.1, construção de circuitos de 2 camadas e o Lema 3.1, técnica telescópica. Os pesos da primeira camada são iguais a 1 e os da segunda camada iguais a 2. O diagrama geral, com entradas, saídas, conexões, pesos e limiares, do multiplicador em $GF(2^8)$ utilizando portas de limiar pode ser visualizado na Figura 4.20.

Figura 4.10 - Multiplicador em $GF(2^8)$ utilizando portas de limiar linear



4.4 Conclusão

Na implementação de projetos de multiplicadores utilizando a lógica tradicional, geralmente é encontrado nas referências circuitos com *fan-in* ilimitado. Isto é justificado para diminuir o tamanho do circuito, já que este cresce exponencialmente com o número de entradas. Como mostrado em [7], é possível, com a utilização de redes neurais discretas e ao tornar a profundidade do circuito ilimitada, reduzir o tamanho dos circuitos de exponencial para polinomial.

Assim, utilizando estas arquiteturas de multiplicadores em corpos finitos com o uso das portas de limiar linear descritas neste capítulo, é possível obter circuitos cujo tamanho não aumenta exponencialmente com o número de entradas, como pode ser observado nas equações 4.13 e 4.27, Quadro 4.1, Figura 4.1, no desenvolvimento do multiplicador $GF(2^4)$ e $GF(2^8)$ e seus respectivos circuitos de limiar lineares.

5. IMPLEMENTAÇÕES DE MULTIPLICADORES EM GF(2^n)

5. Implementações de Multiplicadores em $GF(2^n)$

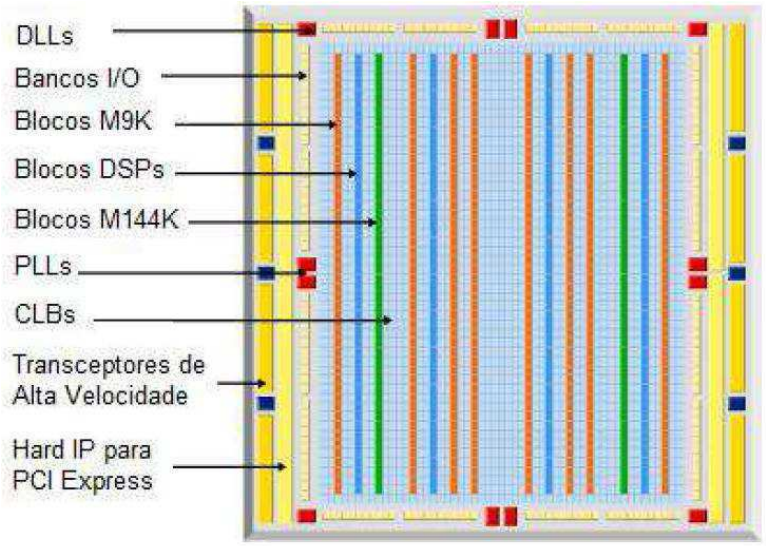
Neste capítulo se concentram a implementação e análise de arquiteturas para unidades aritméticas em corpos finitos de característica 2, em especial a multiplicação corpos finitos em $GF(2^4)$ e $GF(2^8)$. Estas unidades são a base para a implementação de circuitos mais complexos codificadores e decodificadores de canal. A restrição para corpos de característica 2 se deve à natureza binária dos circuitos digitais.

Muitas aplicações exigem a determinação e obtenção de polinômios irredutíveis sobre os Corpos Finitos $GF(q)$. No caso mais simples, $q = 2$, tem-se definido o Corpo Binário $GF(2)$. Em $GF(2)$ a representação dos elementos é bastante simplificada e as operações aritméticas básicas de adição e multiplicação módulo-2 são implementadas a partir de técnicas e estruturas lógicas baseadas na álgebra booleana e álgebra em corpos finitos.

A FPGA (*Field-Programmable Gate Array*) é um dispositivo lógico programável que possui uma arquitetura baseada em blocos lógicos configuráveis, chamados de CLBs (*Configuration Logical Blocks*). Os CLBs são formados por *look-up tables*, multiplexadores e flip-flops que implementam funções lógicas. As FPGAs também são compostas por estruturas chamadas de blocos de entrada e saídas (*IOB Blocks*), os quais são responsáveis pela interface entre as entradas e saídas provenientes das combinações de CLBs.

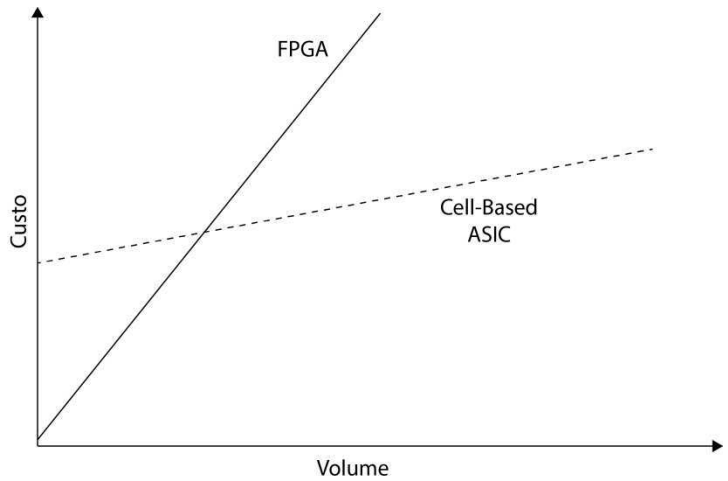
As FPGAs modernas além dos blocos lógicos configuráveis, também tem em sua estrutura blocos de memória RAM (*Random Access Memory*) de alta profundidade, blocos DSP (*Digital Signal Processor*) que realizam operações aritméticas em centenas de megahertz, PLL (*Phase-Locked Loop*) e DLL (*Delay-Locked Loop*) que gerenciam os sinais de relógio e blocos com aplicação específica como interface PCI Express (*Peripheral Component Interconnect Express*) implementado em circuito dedicado. Como exemplo está mostrado na Figura 5.1 [29], a estrutura de uma FPGA da família Stratix do fabricante Altera.

Figura 5.1 - Arquitetura FPGA Stratix



O mercado de FPGA é ocupado em 80% por dois principais fabricantes, que são Xilinx e Altera. As FPGA's também são utilizadas para prototipagem de ASIC's, devido a possibilidade de inúmeras regravações. Elas também são usadas na produção de equipamentos em baixa escala distribuídos em diversas áreas como equipamentos médicos, setor automotivo, comunicações com e sem fio, área militar entre outras. As principais vantagens do uso de FPGA são o alto desempenho, redução do tempo de projeto, possibilidade de reconfiguração do dispositivo depois de implantado e possibilidade de utilização de linguagens com alto nível de abstração para configuração do comportamento do circuito. A principal desvantagem em relação aos ASIC's é o maior custo por unidade, dependendo da demanda, como pode-se observar na Figura 5.2 [30].

Figura 5.2 - Comparação entre Custo e Volume para Tecnologias em FPGA e ASIC [30]



O uso da FPGA, em computação de alto desempenho, está cada vez mais relevante, visto que podem acelerar segmentos críticos em diversas aplicações. Isso tem aumentado o uso deste processamento na implementação sobre corpos finitos em $GF(2^n)$, embora seja uma área que ainda não se tenha avançado muito.

Nas próximas seções apresenta-se diferentes implementações de projetos de multiplicadores em $GF(2^4)$ e $GF(2^8)$, a última delas, na seção 5.4, a implementação e análise de parâmetros do multiplicador em $GF(2^8)$ utilizando redes neurais discretas, objetivo final da dissertação.

5.1 Implementação do Multiplicador de Mastrovito em $GF(2^4)$

Utilizando a Equação (2.23) pode-se escrever a saída do multiplicador de Mastrovito em função das entradas $a_i b_k$ considerando $n = 4$ o tamanho do corpo. Como a saída é uma combinação de portas lógicas AND realizando a operação de multiplicação e a porta lógica XOR realizando a operação de adição, ambas em corpos finitos, tem-se a implementação realizada utilizando circuitos digitais, representada na Figura 5.3 e 5.4.

Figura 5.3 - Implementação em hardware do multiplicador de Mastrovito em $GF(2^4)$

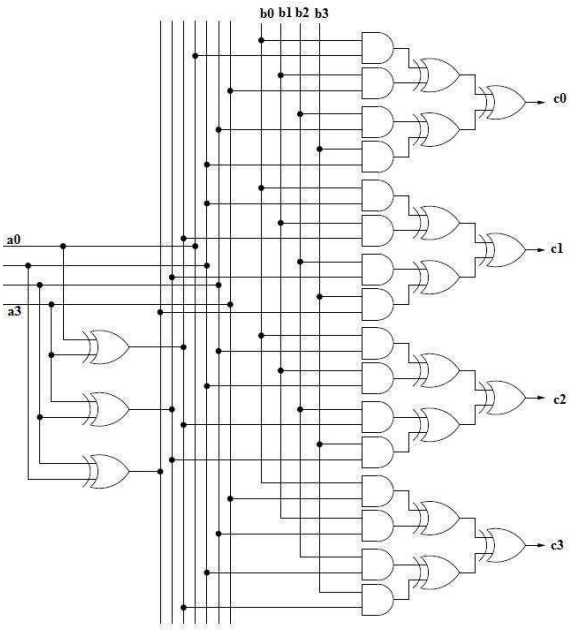
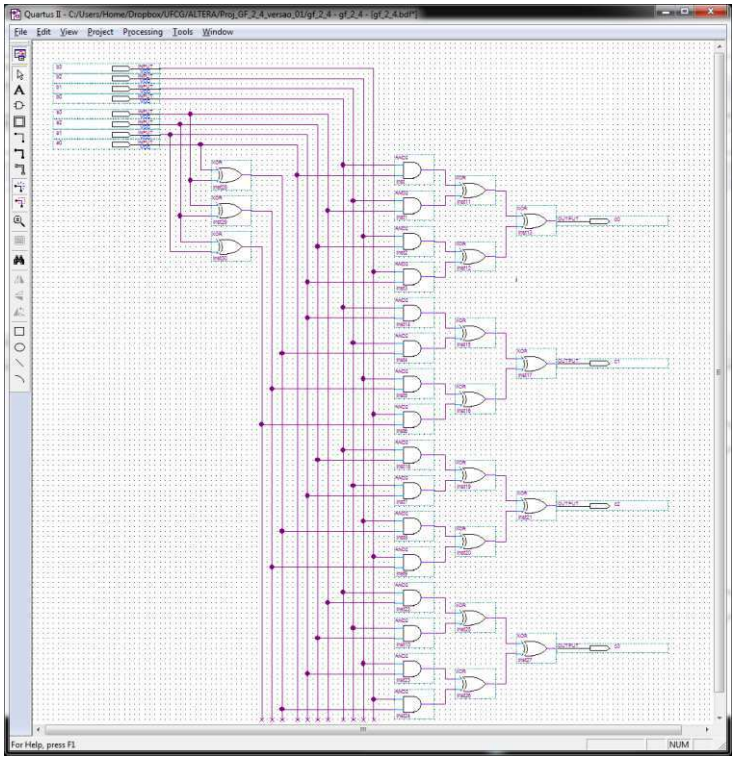


Figura 5.4 - Simulação do multiplicador de Mastrovito em $GF(2^4)$

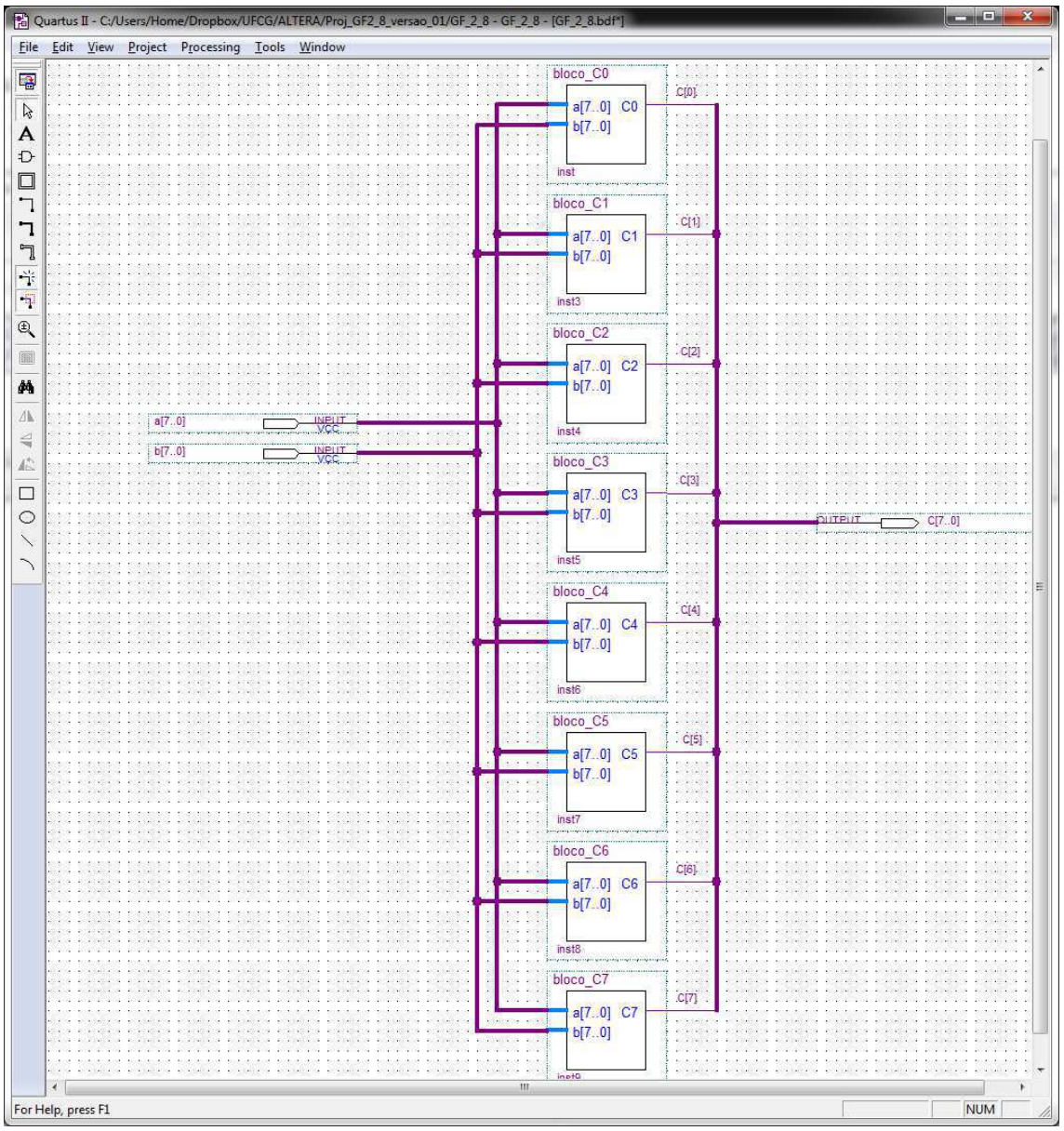


Em termos de portas lógicas utilizadas, o resultado esperado, considerando a complexidade espacial do multiplicador de Mastrovito utilizando um polinômio ótimo, era de 31 portas lógicas. Obtivemos, nesta implementação, o total de 31 portas lógicas (15 portas XOR e 16 portas AND) para realizar a operação.

5.2 Implementação do Multiplicador de Mastrovito $GF(2^8)$

Utilizando a mesma Equação (2.23) podemos escrever a saída do multiplicador de Mastrovito em função das entradas $a_i b_k$ considerando $n = 8$ o tamanho do corpo. Como a saída é uma combinação de portas lógicas AND realizando a operação de multiplicação e a porta lógica XOR realizando a operação de adição, ambas em corpos finitos, tem-se a implementação realizada utilizando circuitos digitais, representada na Figura 5.5.

Figura 5.5 - Simulação do multiplicador de Mastrovito em $GF(2^8)$



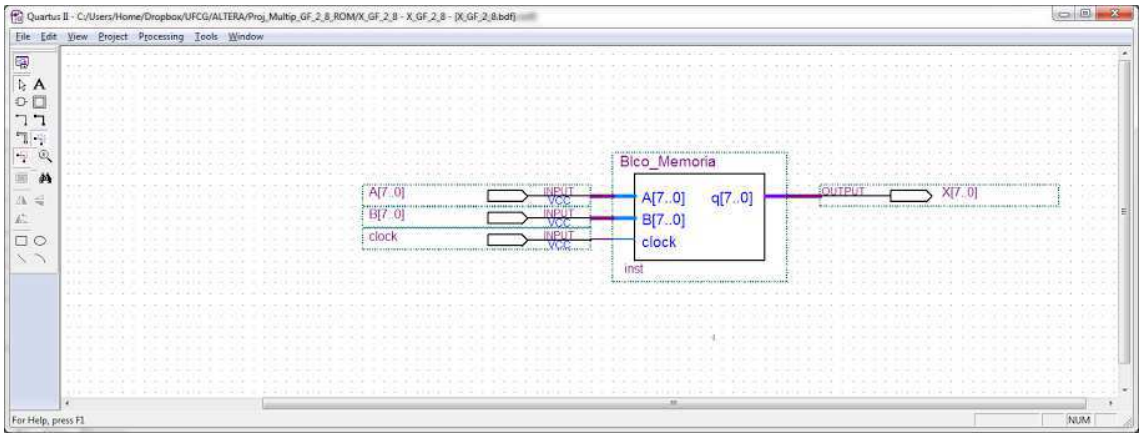
Em termos de portas lógicas utilizadas, o resultado esperado, considerando a complexidade espacial do multiplicador de Mastrovito utilizando o polinômio irredutível $p(x) = x^n + x^{n-1} + \cdots + x + 1$, era de 127 portas lógicas. Obteve-se, nesta implementação, com *clock* de 40 MHz, o total de 141 portas lógicas para realizar a operação.

5.3 Multiplicador em $GF(2^8)$ utilizando células de memória

Utilizando o código descrito na Seção 2.2.1, gera-se uma matriz simétrica com 256 linhas e colunas, totalizando 65536 elementos. A partir desta tábua de multiplicação em corpo finito, insere-se todos os elementos possíveis em diferentes posições de memória do FPGA (*Field Programmable Gate Array*) utilizando o simulador lógico e programável Quartus II, desenvolvido pela Altera.

A partir da seleção de entrada, dada por $a_i b_k$, e do *clock,* em 35 MHz, fez-se a correspondência e busca na matriz simétrica obtida em cada posição de memória. Na Figura 5.6 temos a implementação da multiplicação utilizando as células de memória do dispositivo.

Figura 5.6 - Implementação em *hardware* do multiplicador em $GF(2^8)$ utilizando células de memória.



Estas implementações, utilizando circuitos digitais tradicionais, foram realizadas no sentido de validar a operação, a tábua de multiplicação e de se obter uma maior familiaridade com o projeto do desenvolvimento da nova arquitetura do multiplicador, pois o multiplicador de Mastrovito é utilizado como base para a o novo multiplicador proposto utilizando portas de limiar linear.

5.4 Multiplicador em $GF(2^8)$ utilizando redes neurais discretas

Descrevendo o circuito de limiar linear da Figura 4.20 em Verilog, de acordo com o código descrito no Apêndice I, temos 8 módulos distintos que calculam a paridade de diferentes variáveis de entrada e utilizando limiares previamente calculados

utilizando as técnicas descritas na seção 3.5.1 e Lema 3.1. Para o *bit* de saída c_0, por exemplo, tem-se em linguagem de descrição de *hardware* explicitada no Quadro 5.1.

Quadro 5.1 - Código em Verilog para a implementação do bit c_0

Bit: C0

```
//a0b0; a0b0; a1b7; a2b7; a3b7; a4b7; a5b7; a6b7; a7b7;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER0_IN_LENGTH = 9;
parameter L1_ADDER0_OUT_LENGTH = 4;

wire [L1_ADDER0_OUT_LENGTH-1:0] L1_ADDER0_OUT;

defparam L1_ADDER0.input_length = L1_ADDER0_IN_LENGTH;
defparam L1_ADDER0.output_length = L1_ADDER0_OUT_LENGTH;

bitadder L1_ADDER0
(
	.in
	(
		{
			L0_COMP_OUT[0*GFM_IN_LENGTH+0],
			L0_COMP_OUT[0*GFM_IN_LENGTH+7],
			L0_COMP_OUT[1*GFM_IN_LENGTH+7],
			L0_COMP_OUT[2*GFM_IN_LENGTH+7],
			L0_COMP_OUT[3*GFM_IN_LENGTH+7],
			L0_COMP_OUT[4*GFM_IN_LENGTH+7],
			L0_COMP_OUT[5*GFM_IN_LENGTH+7],
			L0_COMP_OUT[6*GFM_IN_LENGTH+7],
			L0_COMP_OUT[7*GFM_IN_LENGTH+7]
		}
	),
	.out
	(
		{
			L1_ADDER0_OUT
		}
	)
);

//assign LEDR = L1_ADDER0_OUT; // TP LVL1 ADDER0 RESULT

wire [L1_ADDER0_IN_LENGTH/2:0] L1_COMP0;

generate

	genvar L1_COMP0_GEN_I;

	for
	(
		L1_COMP0_GEN_I = 0;
		(L1_COMP0_GEN_I*2+1) <= L1_ADDER0_IN_LENGTH;
		L1_COMP0_GEN_I = L1_COMP0_GEN_I + 1
	)
	begin : L1_COMP0_GENBLOCK
		assign L1_COMP0[ L1_COMP0_GEN_I ] = ( L1_ADDER0_OUT  >=
L1_COMP0_GEN_I*2+1 )?1'b1:1'b0;
	end
```

```
endgenerate

//assign LEDR = L1_COMP0; // TP LVL1 COMP0 RESULT

parameter L2_ADDER0_OUT_LENGTH = L1_ADDER0_OUT_LENGTH;

wire [L2_ADDER0_OUT_LENGTH-1:0] L2_ADDER0_OUT;

bitadder L2_ADDER0
(
      .in(L1_COMP0),
      .out(L2_ADDER0_OUT)
);

wire [L2_ADDER0_OUT_LENGTH-1:0] PARITY_SUB0 =  {L2_ADDER0_OUT, 1'b0}-
L1_ADDER0_OUT;

assign GFM_OUT[0] = ( PARITY_SUB0 >= 1 )?1'b1:1'b0;
```

Foram implementados no total 8 módulos, em que cada um deles corresponde a diferentes saídas, compondo o vetor $c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7$. Cada módulo calcula a paridade da soma das variáveis de entrada através das comparações com os limiares e multiplicações pelos pesos de cada conexão. Para todas as possibilidades de soma das variáveis nas entradas de cada módulo foram testados o funcionamento do cálculo da função paridade, como pode ser visualizado na verificação a seguir. A variável IN são as possibilidades de entradas, PC0 fornece o somatório dos *bits* de entrada, COMP0 o valor limiar com o qual será realizada a comparação, PC1 a soma dos *bits* de COMP e OUT a saída da realização da paridade. No Quadro 5.2 tem-se as possibilidades de entrada do circuito de limiar, que varia de 8 a 13 variáveis, e o resultado da verificação da paridade utilizando a estrutura descrita na Figura 4.20.

Quadro 5.2 - Verificação da Paridade

```
  IN: 0000000000000; PC0:  0; COMP0: 0000000000000; PC1:  0; OUT: 0;
  IN: 0000000000001; PC0:  1; COMP0: 0000000000001; PC1:  0; OUT: 1;
  IN: 0000000000011; PC0:  2; COMP0: 0000000000001; PC1:  1; OUT: 0;
  IN: 0000000000111; PC0:  3; COMP0: 0000000000011; PC1:  2; OUT: 1;
  IN: 0000000001111; PC0:  4; COMP0: 0000000000011; PC1:  2; OUT: 0;
  IN: 0000000011111; PC0:  5; COMP0: 0000000000111; PC1:  2; OUT: 1;
  IN: 0000000111111; PC0:  6; COMP0: 0000000000111; PC1:  3; OUT: 0;
  IN: 0000001111111; PC0:  7; COMP0: 0000000001111; PC1:  3; OUT: 1;
  IN: 0000011111111; PC0:  8; COMP0: 0000000001111; PC1:  4; OUT: 0;
  IN: 0000111111111; PC0:  9; COMP0: 0000000011111; PC1:  4; OUT: 1;
  IN: 0001111111111; PC0: 10; COMP0: 0000000011111; PC1:  5; OUT: 0;
  IN: 0011111111111; PC0: 11; COMP0: 0000000111111; PC1:  5; OUT: 1;
  IN: 0111111111111; PC0: 12; COMP0: 0000000111111; PC1:  6; OUT: 0;
  IN: 1111111111111; PC0: 13; COMP0: 0000000111111; PC1:  6; OUT: 1;
- NeuralParityCheck_TestBench.v:14: Verilog $finish
- NeuralParityCheck_TestBench.v:14: Verilog $finish
- NeuralParityCheck_TestBench.v:14: Second verilog $finish, exiting
```

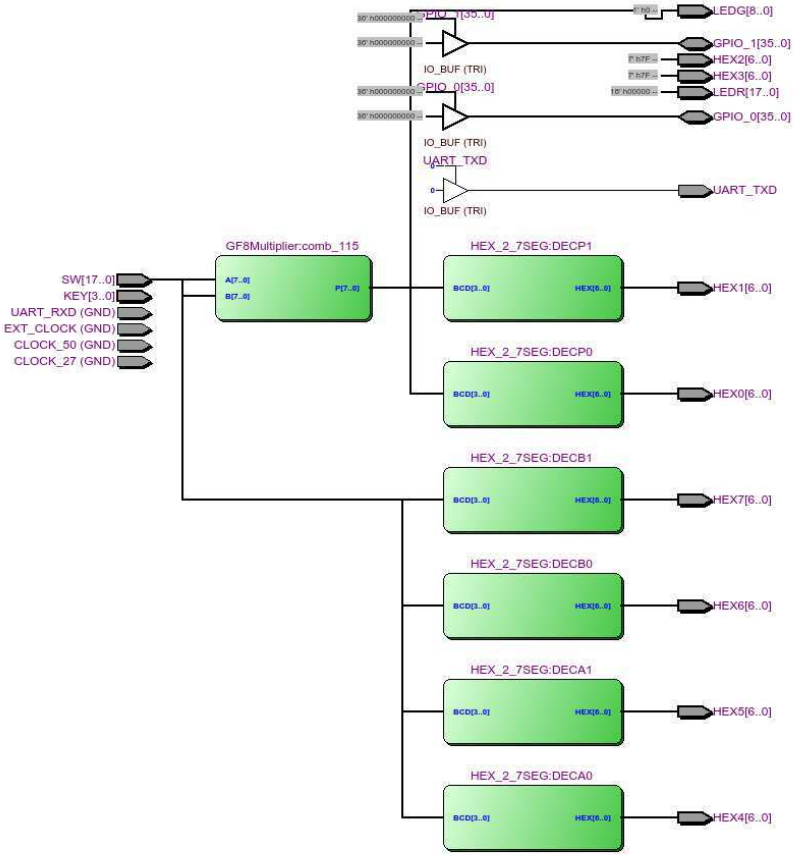
A frequência de operação da FPGA é de 50 MHz. O menor atraso de propagação, tempo mínimo entre a propagação do dado da entrada para a saída, foi de 2,640 ns, da chave SW[10] para o HEX6[0]. O maior atraso de propagação obtido foi de 9,609 ns. No Apêndice III temos os atrasos mínimos de propagação para todas as entradas e saídas.

O *software* de desenvolvimento Quartus II® é a ferramenta da *Altera Corporation*® [30] para a síntese lógica de projetos digitais, que suporta toda a sua linha de dispositivos lógicos programáveis (FPGA's), e que também permite o projeto de ASIC's. O Quartus II® integra diversas funções, tais como:

- Entrada e edição de projetos através de diagramas de bloco, AHDL, VHDL e Verilog HDL;
- Combinação de diferentes tipos de arquivos de projeto;
- Simulação funcional e temporizada;
- Localização automática de erros;
- Funções de localização e roteamento, verificação e programação de dispositivos;
- Análise e cálculo dos tempos de propagação;
- Capacidade de otimização automática do consumo de energia do projeto;
- Criação e otimização de blocos de projeto de forma independente.

Das diversas funcionalidades apresentadas, utilizamos uma ferramenta que exibe um diagrama esquemático do circuito projetado, o *RTL Viewer*, que fornece a estrutura do código em nível de transferência de registrador. RTL é o nível de abstração mais alto usado no fluxo tradicional de projeto de sistemas digitais. Na Figura 5.7 temos o maior nível de abstração do multiplicador em $GF(2^8)$, gerado a partir da descrição de *hardware* do Apêndice I.

Figura 5.7 - *RTL Viewer* do Multiplicador em $GF(2^8)$ utilizando portas de limiar linear



O bloco *GF8Multiplier:comb_115* é o bloco principal, onde toda a descrição do circuito de limiar é aplicada. Os demais níveis de abstração podem ser visualizados no Apêndice IV.

Para a multiplicação em $GF(2^8)$ utilizando portas de limiar implementada em FPGA, obteve-se um total de 296 portas lógicas necessárias para computar a operação, menos que 1% da capacidade do dispositivo EP2C35F672C6. A potência fornecida pela ferramenta de síntese foi de 132,58 mW. Parâmetros como o total de portas utilizadas, *fan-out*, máximo *fan-out*, potência térmica total, correntes necessárias para o cálculo de cada coeficiente c_n utilizando a função paridade, tensão de operação e corrente total na implementação são mostradas nos Quadros 5.3 e 5.4. Detalhes adicionais dos resultados gerados na síntese do Quartus II® podem ser encontrados no Apêndice III.

Quadro 5.3 - Complexidade Espacial, *fan-out* e correntes na implementação do multiplicador em GF(2^8)

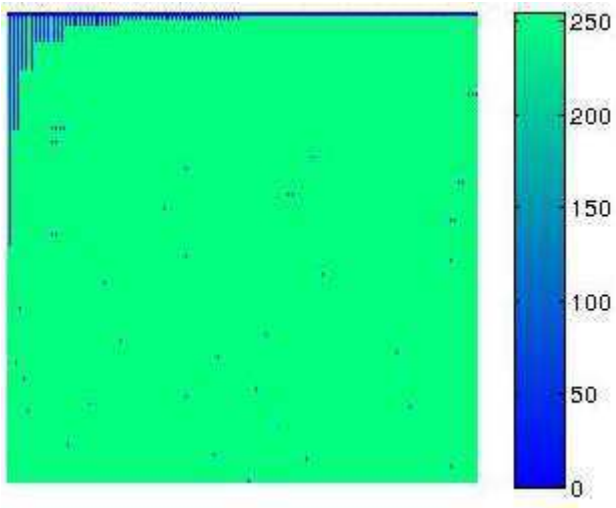
Total de Portas Utilizadas	***Fan-out***	**Máximo *Fan-out***	**Potência Total**	**Corrente I/O**	**Corrente Operação**
296	1090	44	132,58 mW	11,39 mA	79,16 mA

Quadro 5.4 - Tensão e Correntes de operação por bit de saída do multiplicador em GF(2^8)

I/O	**Tensão (V)**	**Corrente (mA)**
1	3,3 V	1,44 mA
2	3,3 V	1,32 mA
3	3,3 V	1,15 mA
4	3,3 V	1,15 mA
5	3,3 V	1,82 mA
6	3,3 V	1,79 mA
7	3,3 V	1,44 mA
8	3,3 V	1,30 mA

A partir dos resultados experimentais obtidos e da tábula de multiplicação em $GF(2^8)$ obtida com o algoritmo no Matlab® na seção 2.2.1, fez-se um mapa de verificação da operação com a finalidade de comparar o resultado da multiplicação teórico e o obtido com a implementação na FPGA utilizando portas de limiar linear. No Apêndice II tem-se o algoritmo que fornece a diferença entre os resultados e este mapeamento. Na Figura 5.8 pode-se ver esta comparação pelo mapa de verificação, em que a cor verde, região mais uniforme, representa a operação realizada corretamente e a cor azul a representação de uma falha na operação. Para o cálculo realizado, obteve-se uma taxa de acerto de aproximadamente 90%. Portanto, faz-se necessária uma depuração do projeto algébrico e uma análise sobre o tempo de processamento, pois algumas computações podem estar sendo realizadas antes que os *bits* estejam disponíveis, de modo que este erro seja eliminado.

Figura 5.8 - Mapa de Verificação do Multiplicador em $GF(2^8)$



5.5 Conclusões

Nesse capítulo foram descritos os passos desenvolvidos para a implementação do multiplicador em $GF(2^8)$ utilizando portas de limiar linear como elemento básico de processamento em uma rede neural discreta que realiza o cálculo da função paridade. Também foram apresentados os resultados das simulações de diferentes implementações, desde circuitos digitais convencionais, circuitos em FPGA utilizando células de memória e implementação, objeto de final de estudo, do multiplicador em $GF(2^8)$ utilizando portas de limiar linear. Com isso, o objetivo de comprovar o funcionamento prático da arquitetura do multiplicador em $GF(2^8)$ foi cumprido. Entretanto, sua eficiência, em termos de tempo de processamento e contagem de portas está abaixo do esperado. Em termos de resultados da operação de multiplicação em corpos finitos, observa-se uma taxa de acerto de 90%.

5. CONSIDERAÇÕES FINAIS E PERSPECTIVAS

6. Considerações Finais e Perspectivas

Neste trabalho foram apresentadas diferentes implementações em *hardware* para multiplicadores em corpos finitos utilizando portas de limiar linear. Devido as potencialidades destas portas foram obtidos resultados satisfatórios. Para a nova arquitetura proposta foram previstas um total de 120 portas para realizar a operação. Na implementação obtivemos um total de 296 portas, com a ressalva que este número pode ser reduzido com a otimização da descrição em HDL Verilog. No entanto, este valor é menor que o número total de portas do multiplicador tradicional, 493. Portanto, obteve-se uma redução de 40% das portas. O tempo de atraso da operação, de 2,0 a 10 ns, foi similar ao encontrado nas implementações em outras referências de multiplicadores paralelos de alto desempenho, como mostrado em [31-37]. A porta de limiar linear é o elemento básico de uma rede neural discreta, cuja utilização visa diminuir a complexidade espacial e temporal dos circuitos para multiplicação em corpos finitos. Para a realização do projeto, foi necessário um estudo dos conceitos relacionados à multiplicação em corpos finitos, bem como as arquiteturas existentes para implementação em *hardware*. Desse modo, o projeto e implementação de multiplicadores em corpos finitos, que utilizam portas de limiar como elemento de processamento, se torna bem promissor.

A seguir são feitas algumas sugestões para a continuação das atividades de pesquisa e alguns trabalhos publicados.

- Utilizar métodos computacionais para determinação de polinômios $p(x)$ de forma que se tenha uma redução da complexidade espacial;
- Busca por um método geral para geração automática do diagrama do multiplicador para qualquer corpo n na base $GF(2)$ para polinômios irredutíveis diferentes;
- Projetar o multiplicador em GF(2^8) utilizando 3 camadas;
- Analisar o tempo de processamento dos multiplicadores implementados neste trabalho;
- Implementação analógica e comparativo, em termos de contagem de portas, potência, atraso e custo, com a implementação digital;

- Projetar um multiplicador maior, como o GF(2^{233}), multiplicador padrão proposto pela NIST (*National Institute of Standards and Technology*);
- Implementação em ASIC do modelo proposto neste trabalho.

Santos M. A., Freire R. C. S, Assis F. M, Reis V. L., Albuquerque T. C. "*Implementação em Hardware do Multiplicador de Mastrovito em GF (2^4)*". Encom2015. Campina Grande, 2015.

Santos M. A., Freire R. C. S, Assis, F. M, Reis, V. L., Araújo A. F. "*Multiplicador em GF (2^8) em FPGA utilizando células de memória*". VIII Colóquio de Matemática. Campina Grande, 2015.

Santos M. A., Freire R. C. S, Assis, F. M, Reis, V. L., Albuquerque T. C. "*GF (2^8) Multiplier on Hardware using Discrete Neural Network*".IsCAS2016. Montreal, 2015.

Referências

[1] Lidiano A. N. Oliveira. “Multiplicador em corpo finito utilizando redes neurais discretas,” Dissertação de Mestrado, Universidade Federal de Campina Grande, Campina Grande, PB, Brasil, Outubro de 2000.

[2] Cristóvão M. de O. Lima F. “Circuito Integrado para Multiplicação em GF(2^4) Utilizando Portas de Limiar Linear" Dissertação de Mestrado, Universidade Federal de Campina Grande, Campina Grande-PB, Brasil, 2010.

[3] Hasan M. A.. “Look-up table-based large finite field multiplication in memory constrained cryptosystems,” IEEE Transactions on Computers, vol. 49, no. 7, pp. 749–758, July 2000.

[4] Imaña J. L., Sánchez J. M. “Multiplicadores canônicos sobre campos GF(2^m) generados por um tipo de trinômios irreducibles implementados em hardware reconfigurable,” III Jornadas sobre Computacion Reconfigurable y Aplicaciones – JCRA2003, pp. 65–72, 2003.

[5] Edgar F., Bollman D., Moreno O. “A fast finite field multiplier,” Reconfigurable Computing: Architectures , Tools and Applications, Third International Workshop, ARC 2007, Mangaratiba, Brazil, pp. 238–246, March 2007.

[6] Ashkan H. Namin, Huapeng Wu, Majid Ahmadi. “High speed word-parallel bitserial normal basis finite field multiplier and its FPGA implementation,” Proc. 39th Asilomar Conf. Signals, Systems, and Computers, pp. 1338–1341, November 2005.

[7] Kailath, Siu and Roychaudhury “Discrete Neural Computation, a Thearetical Foundation” MC-Gral-Hill, 1995.

[8] Todd K. Moon. “Error Correction Coding: Mathematical methods and algoritms,” John Wiley and Sons, New Jersey, 2005.

[9] Paar, Chistof. “A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields”, IEEE Transaction on Computers, Vol 45, No. 7, July 1996.

[10] Peter Celinski, Sorin D. Cotofana, and Derek Abbot, “Threshold logic parallel counters for 32-bit multipliers,” Proceedings of SPIE, vol. 4935, pp. 205–214, 2002.

[11] Y. Leblebici, H. Özdemir, A. Kepkep, and U. Çilingiroglu. “A compact high-speed (31,5) parallel counter circuit based on capacitive threshold-logic gates,” IEEE Journal of Solid-State Circuits, vol. SC-31, pp. 1177–1183, 1996.

[12] David W. Ash, Ian F. Blake, e Scott A. Vanstone. “Low Complexity Normal Bases. In Discrete Applied Mathematics,” volume 25, pp. 191–210. North-Holland: Elsevier Science Publishers, 1989.

[13] Richard E. Blahut. “Theory and Practice of Error Control Codes.” Reading, MA: Addison-Wesley, 1983.

[14] Rudolf Lidl; Harald Niederreiter. “Finite Fields. In Encyclopedia of Mathematics and Its Applications.” Cambridge: University Press, 1997.

[15] Shu Lin; Daniel J. Costello. Error Control Coding: “Fundamentals and Applications.” Englewood Cliffs, NJ: Prentice-Hall, 1983.

[16] Alfred J. Menezes, editor. “Applications of Finite Fields.” Boston: Kluwer Academic Publishers, 1993.

[17] Leocarlos Bezerra da Silva Lima. “Architecture de décodage pour codes algébriques géométriques basés sur des courbes d’Hermite”. Télécom ParisTech, 2004.

[18] S. Lin; D. J. Costello. “Error Control Coding: Fundamentals and Applications. “Prentice-Hall International, Inc., May 1983.

[19] Mastrovito E. D. “VLSI architectures for computation in Galois field,” PhD thesis, Linkoping Univ., Linkoping, Sweden, 1991.

[20] Oliveira, L. A. N. ; Francisco Assis . “A New Architecture for Multipliers in GF(2^n) Using Discrete Neural Networks.” In: III Congresso Brasileiro de Redes Neurais, 1997, Florianopólis. Anais do III Congresso Brasileiro de Redes Neurais, 1997. v. 1. p. 12-15.

[21] Halbutogullari A. and Koc C . K.. “Mastrovito Multiplier for General Irreducible Polynomials". IEEE Transactions on Computers, Vol. 49, No. 5, pp. 503{518, May 2000.

[22] S. Lin and D. J. Costello. “Error Control Coding: Fundamentals and Applications.” Prentice-Hall International, Inc., May 1983.

[23] Shibata T. and Ohmi T., “An intelligent MOS transistor featuring gate-level weighted sum and threshold operation,” IEDM, Technical Digest, IEEE, New York, December 1991.

[24] Padure M., Cotofana S., and Vassiliadis S., “A low-power threshold logic family,” in Proc. IEEE International Conference on Electronics, Circuits and Systems, pp. 657– 66, 2002.

[25] Celinski P.,. Cotofana S. D.,. López J. F, l-Sarawi S., Abbott D., “State-of-art in CMOS threshold-logic VLSI gate implementations and applications,” Proceedings of the VLSI Circuits and Systems Conference, vol. 5117, Spain, pp. 53–64, 2003.

[26] Bruck J. and Smolensky R.. “Polynomial Threshold Functions, AC 0 Functions and Spectral Norms". In Proc. 31st IEEE Symposium on Foundations of Computer Science, Vol. 44, pp. 632{641, February 1990.

[27] Paterson M., Pippenger N., and U. Zwick, “Faster circuits and shorter formulae for multiple addition, multiplication and symmetric Boolean functions,” In Proceedings of the 31st Annual Symposium on Foundations of Computer Science, pp. 642-650, 1990.

[28] Paturi R. and Saks M., “On threshold circuits for parity,” In Proceedings of the 31st Annual Symposium on Foundations of Computer Science, pp. 397-404, 1990.

[29] Altera. About Stratix Series High-end FPGAs. 2010.

[30] Flynn Michael M.J. , Luk W., Computer System Design: System-on-Chip, August 2011, Wiley.

[31] Rong-Jian Chen, Jhen-Wun Fan and Chin-Hao Liao, “Reconfigurable Galois Multiplier”.International Symposium on Biometrics and Security Technologies (ISBAST), 2014.

[32] Retheesh. D, “Analysis on FPGA Designs of Parallel High Performance Multipliers”. International Journal of Communication and Computer Technologies Volume 02 – No.12, 2014.

[33] Garcia-Martinez M.A, Posada-Gomez R., Morales-Luna G., Rodriguez-Henriquez F. “FPGA implementation of an efficient multiplier over finite fields GF(2m)”. International Conference on Reconfigurable Computing and FPGAs. 2005.

[34] P. Kitsos, G. Theodoridis, O. Koufopavlou. “An efficient reconfigurable multiplier for Galois Field GF(2m)”. Microelectronics Journal. Vol 34, pp 975-980. 2003.

[35] Lakhendra K., Sudha K.L., “Implementation of Galois Field Arithmetic Unit on FPGA”, International Journal of Innovative Research in Computes and Communication Engineering, Vol. 2, Issue 6, June 2014.

[36] Maheswari M. U., S. Baskar, G.M. Keerthi, “High Speed Finete Field Multiplier GF(2^m) for Cryptographic Applications”. International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE) Volume 3, Issue 11, November 2014.

[37] Ajitha S.S., D. Retheesh.D, “Efficient Implementation of bit parallel finie fild multipliers”. IJRET: International Journal of Research in Engineering and Technology, 2015.

Apêndice I

Este apêndice tem por objetivo descrever explicitamente os códigos dos módulos e *Test Bench* da descrição em Verilog referente ao cálculo da paridade de funções com 8, 9, 10, 11, 12 e 13 variáveis de entrada projetadas para composição do multiplicador em $GF(2^8)$, como na Equação 4.30 e Figura 4.20 do Capítulo 4 do texto,

Saída: C0

```
//a0b0; a0b0; a1b7; a2b7; a3b7; a4b7; a5b7; a6b7; a7b7;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER0_IN_LENGTH = 9;
parameter L1_ADDER0_OUT_LENGTH = 4;

wire [L1_ADDER0_OUT_LENGTH-1:0] L1_ADDER0_OUT;

defparam L1_ADDER0.input_length = L1_ADDER0_IN_LENGTH;
defparam L1_ADDER0.output_length = L1_ADDER0_OUT_LENGTH;

bitadder L1_ADDER0
(
	.in
	(
		{
			L0_COMP_OUT[0*GFM_IN_LENGTH+0],
			L0_COMP_OUT[0*GFM_IN_LENGTH+7],
			L0_COMP_OUT[1*GFM_IN_LENGTH+7],
			L0_COMP_OUT[2*GFM_IN_LENGTH+7],
			L0_COMP_OUT[3*GFM_IN_LENGTH+7],
			L0_COMP_OUT[4*GFM_IN_LENGTH+7],
			L0_COMP_OUT[5*GFM_IN_LENGTH+7],
			L0_COMP_OUT[6*GFM_IN_LENGTH+7],
			L0_COMP_OUT[7*GFM_IN_LENGTH+7]
		}
	),
	.out
	(
		{
			L1_ADDER0_OUT
		}
	)
);

//assign LEDR = L1_ADDER0_OUT; // TP LVL1 ADDER0 RESULT

wire [L1_ADDER0_IN_LENGTH/2:0] L1_COMP0;

generate

	genvar L1_COMP0_GEN_I;

	for
	(
		L1_COMP0_GEN_I = 0;
		(L1_COMP0_GEN_I*2+1) <= L1_ADDER0_IN_LENGTH;
```

```
            L1_COMP0_GEN_I = L1_COMP0_GEN_I + 1
      )
      begin : L1_COMP0_GENBLOCK
            assign L1_COMP0[ L1_COMP0_GEN_I ] = ( L1_ADDER0_OUT  >=
L1_COMP0_GEN_I*2+1 )?1'b1:1'b0;
      end

endgenerate

//assign LEDR = L1_COMP0; // TP LVL1 COMP0 RESULT

parameter L2_ADDER0_OUT_LENGTH = L1_ADDER0_OUT_LENGTH;

wire [L2_ADDER0_OUT_LENGTH-1:0] L2_ADDER0_OUT;

bitadder L2_ADDER0
(
      .in(L1_COMP0),
      .out(L2_ADDER0_OUT)
);

wire [L2_ADDER0_OUT_LENGTH-1:0] PARITY_SUB0 =  {L2_ADDER0_OUT, 1'b0}-
L1_ADDER0_OUT;

assign GFM_OUT[0] = ( PARITY_SUB0 >= 1 )?1'b1:1'b0;
```

Saída: C1

```
//a1b0; a1b7; a2b6; a3b5; a4b4; a5b3; a6b2; a7b1;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER1_IN_LENGTH = 9;
parameter L1_ADDER1_OUT_LENGTH = 4;

wire [L1_ADDER1_OUT_LENGTH-1:0] L1_ADDER1_OUT;

defparam L1_ADDER1.input_length = L1_ADDER1_IN_LENGTH;
defparam L1_ADDER1.output_length = L1_ADDER1_OUT_LENGTH;

bitadder L1_ADDER1
(
      .in
      (
            {
                  L0_COMP_OUT[1*GFM_IN_LENGTH+0],
                  L0_COMP_OUT[1*GFM_IN_LENGTH+7],
                  L0_COMP_OUT[2*GFM_IN_LENGTH+6],
                  L0_COMP_OUT[3*GFM_IN_LENGTH+5],
                  L0_COMP_OUT[4*GFM_IN_LENGTH+4],
                  L0_COMP_OUT[5*GFM_IN_LENGTH+3],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+2],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+1]
             }

      ),
      .out
      (
            {
```

```
                  L1_ADDER1_OUT
            }
      )
);

//assign LEDR = L1_ADDER1_OUT; // TP LVL1 ADDER1 RESULT

wire [L1_ADDER1_IN_LENGTH/2:0] L1_COMP1;

generate

      genvar L1_COMP1_GEN_I;

      for
      (
            L1_COMP1_GEN_I = 0;
            (L1_COMP1_GEN_I*2+1) <= L1_ADDER1_IN_LENGTH;
            L1_COMP1_GEN_I = L1_COMP1_GEN_I + 1
      )
      begin : L1_COMP1_GENBLOCK
            assign L1_COMP1[ L1_COMP1_GEN_I ] = ( L1_ADDER1_OUT  >=
L1_COMP1_GEN_I*2+1 )?1'b1:1'b0;
      end

endgenerate

//assign LEDR = L1_COMP1; // TP LVL1 COMP1 RESULT

parameter L2_ADDER1_OUT_LENGTH = L1_ADDER1_OUT_LENGTH;

wire [L2_ADDER1_OUT_LENGTH-1:0] L2_ADDER1_OUT;

bitadder L2_ADDER1
(
      .in(L1_COMP1),
      .out(L2_ADDER1_OUT)
);

wire [L2_ADDER1_OUT_LENGTH-1:0] PARITY_SUB1 =  {L2_ADDER1_OUT, 1'b0}-
L1_ADDER1_OUT;

assign GFM_OUT[1] = ( PARITY_SUB1 >= 1 )?1'b1:1'b0;
```

Saída:C2

```
// a1b1; a2b0; a2b7; a3b6; a3b7; a4b5; a4b6; a5b4; a5b5; a6b3; a6b4;
a7b2; a7b3
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER2_IN_LENGTH = 9;
parameter L1_ADDER2_OUT_LENGTH = 4;

wire [L1_ADDER2_OUT_LENGTH-1:0] L1_ADDER2_OUT;

defparam L1_ADDER2.input_length = L1_ADDER2_IN_LENGTH;
defparam L1_ADDER2.output_length = L1_ADDER2_OUT_LENGTH;

bitadder L1_ADDER2
(
      .in
```

```
	(
		{

			L0_COMP_OUT[1*GFM_IN_LENGTH+1],
			L0_COMP_OUT[2*GFM_IN_LENGTH+0],
			L0_COMP_OUT[2*GFM_IN_LENGTH+7],
			L0_COMP_OUT[3*GFM_IN_LENGTH+6],
			L0_COMP_OUT[3*GFM_IN_LENGTH+7],
			L0_COMP_OUT[4*GFM_IN_LENGTH+5],
			L0_COMP_OUT[4*GFM_IN_LENGTH+6],
			L0_COMP_OUT[5*GFM_IN_LENGTH+4],
			L0_COMP_OUT[5*GFM_IN_LENGTH+5],
			L0_COMP_OUT[6*GFM_IN_LENGTH+3],
			L0_COMP_OUT[6*GFM_IN_LENGTH+4],
			L0_COMP_OUT[7*GFM_IN_LENGTH+2],
			L0_COMP_OUT[7*GFM_IN_LENGTH+3]
		}
	),
	.out
	(
		{
			L1_ADDER2_OUT
		}
	)
);

//assign LEDR = L1_ADDER2_OUT; // TP LVL1 ADDER2 RESULT

wire [L1_ADDER2_IN_LENGTH/2:0] L1_COMP2;

generate

	genvar L1_COMP2_GEN_I;

	for
	(
		L1_COMP2_GEN_I = 0;
		(L1_COMP2_GEN_I*2+1) <= L1_ADDER2_IN_LENGTH;
		L1_COMP2_GEN_I = L1_COMP2_GEN_I + 1
	)
	begin : L1_COMP2_GENBLOCK
		assign L1_COMP2[ L1_COMP2_GEN_I ] = ( L1_ADDER2_OUT  >=
L1_COMP2_GEN_I*2+1 )?1'b1:1'b0;
	end

endgenerate

//assign LEDR = L1_COMP2; // TP LVL1 COMP2 RESULT

parameter L2_ADDER2_OUT_LENGTH = L1_ADDER2_OUT_LENGTH;

wire [L2_ADDER2_OUT_LENGTH-1:0] L2_ADDER2_OUT;

bitadder L2_ADDER2
(
	.in(L1_COMP2),
	.out(L2_ADDER2_OUT)
);

wire [L2_ADDER2_OUT_LENGTH-1:0] PARITY_SUB2 =  {L2_ADDER2_OUT, 1'b0}-
L1_ADDER2_OUT;
```

```
assign GFM_OUT[2] = ( PARITY_SUB2 >= 1 )?1'b1:1'b0;
```

Saída: C3

```
//a1b2;a2b1;a3b0;a3b7;a4b6;a4b7;a5b5;a5b6;a6b4;a6b5;a7b3;a7b4;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER3_IN_LENGTH = 9;
parameter L1_ADDER3_OUT_LENGTH = 4;

wire [L1_ADDER3_OUT_LENGTH-1:0] L1_ADDER3_OUT;

defparam L1_ADDER3.input_length = L1_ADDER3_IN_LENGTH;
defparam L1_ADDER3.output_length = L1_ADDER3_OUT_LENGTH;

bitadder L1_ADDER3
(
	.in
	(
		{
			L0_COMP_OUT[1*GFM_IN_LENGTH+2],
			L0_COMP_OUT[2*GFM_IN_LENGTH+1],
			L0_COMP_OUT[3*GFM_IN_LENGTH+0],
			L0_COMP_OUT[3*GFM_IN_LENGTH+7],
			L0_COMP_OUT[4*GFM_IN_LENGTH+6],
			L0_COMP_OUT[4*GFM_IN_LENGTH+7],
			L0_COMP_OUT[5*GFM_IN_LENGTH+5],
			L0_COMP_OUT[5*GFM_IN_LENGTH+6],
			L0_COMP_OUT[6*GFM_IN_LENGTH+4],
			L0_COMP_OUT[6*GFM_IN_LENGTH+5],
			L0_COMP_OUT[7*GFM_IN_LENGTH+3],
			L0_COMP_OUT[7*GFM_IN_LENGTH+4]

		}
	),
	.out
	(
		{
			L1_ADDER3_OUT
		}
	)
);

//assign LEDR = L1_ADDER3_OUT; // TP LVL1 ADDER3 RESULT

wire [L1_ADDER3_IN_LENGTH/2:0] L1_COMP3;

generate

	genvar L1_COMP3_GEN_I;

	for
	(
		L1_COMP3_GEN_I = 0;
		(L1_COMP3_GEN_I*2+1) <= L1_ADDER3_IN_LENGTH;
		L1_COMP3_GEN_I = L1_COMP3_GEN_I + 1
```

```
      )
      begin : L1_COMP3_GENBLOCK
            assign L1_COMP3[ L1_COMP3_GEN_I ] = ( L1_ADDER3_OUT  >=
L1_COMP3_GEN_I*2+1 )?1'b1:1'b0;
      end

endgenerate

//assign LEDR = L1_COMP3; // TP LVL1 COMP3 RESULT

parameter L2_ADDER3_OUT_LENGTH = L1_ADDER3_OUT_LENGTH;

wire [L2_ADDER3_OUT_LENGTH-1:0] L2_ADDER3_OUT;

bitadder L2_ADDER3
(
      .in(L1_COMP3),
      .out(L2_ADDER3_OUT)
);

wire [L2_ADDER3_OUT_LENGTH-1:0] PARITY_SUB3 =  {L2_ADDER3_OUT, 1'b0}-
L1_ADDER3_OUT;

assign GFM_OUT[3] = ( PARITY_SUB3 >= 1 )?1'b1:1'b0;
```

Saída C4:

```
//a1b3; a2b2; a3b1; a4b0; a4b7; a5b6; a5b7; a6b5; a6b6; a7b4; a7b5;

//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER4_IN_LENGTH = 9;
parameter L1_ADDER4_OUT_LENGTH = 4;

wire [L1_ADDER4_OUT_LENGTH-1:0] L1_ADDER4_OUT;

defparam L1_ADDER4.input_length = L1_ADDER4_IN_LENGTH;
defparam L1_ADDER4.output_length = L1_ADDER4_OUT_LENGTH;

bitadder L1_ADDER4
(
      .in
      (
            {
                  L0_COMP_OUT[1*GFM_IN_LENGTH+3],
                  L0_COMP_OUT[2*GFM_IN_LENGTH+2],
                  L0_COMP_OUT[3*GFM_IN_LENGTH+1],
                  L0_COMP_OUT[4*GFM_IN_LENGTH+0],
                  L0_COMP_OUT[4*GFM_IN_LENGTH+7],
                  L0_COMP_OUT[5*GFM_IN_LENGTH+6],
                  L0_COMP_OUT[5*GFM_IN_LENGTH+7],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+5],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+6],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+4],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+5]

            }
```

```
	),
	.out
	(
		{
			L1_ADDER4_OUT
		}
	)
);

//assign LEDR = L1_ADDER4_OUT; // TP LVL1 ADDER4 RESULT

wire [L1_ADDER4_IN_LENGTH/2:0] L1_COMP4;

generate

	genvar L1_COMP4_GEN_I;

	for
	(
		L1_COMP4_GEN_I = 0;
		(L1_COMP4_GEN_I*2+1) <= L1_ADDER4_IN_LENGTH;
		L1_COMP4_GEN_I = L1_COMP4_GEN_I + 1
	)
	begin : L1_COMP4_GENBLOCK
		assign L1_COMP4[ L1_COMP4_GEN_I ] = ( L1_ADDER4_OUT  >=
L1_COMP4_GEN_I*2+1 )?1'b1:1'b0;
	end

endgenerate

//assign LEDR = L1_COMP4; // TP LVL1 COMP4 RESULT

parameter L2_ADDER4_OUT_LENGTH = L1_ADDER4_OUT_LENGTH;

wire [L2_ADDER4_OUT_LENGTH-1:0] L2_ADDER4_OUT;

bitadder L2_ADDER4
(
	.in(L1_COMP4),
	.out(L2_ADDER4_OUT)
);

wire [L2_ADDER4_OUT_LENGTH-1:0] PARITY_SUB4 =  {L2_ADDER4_OUT, 1'b0}-
L1_ADDER4_OUT;

assign GFM_OUT[4] = ( PARITY_SUB4 >= 1 )?1'b1:1'b0;
```

Saída: C5

```
//a1b4; a2b3; a3b2; a4b1; a5b0; a5b7; a6b6; a6b7; a7b5; a7b6;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER5_IN_LENGTH = 9;
parameter L1_ADDER5_OUT_LENGTH = 4;

wire [L1_ADDER5_OUT_LENGTH-1:0] L1_ADDER5_OUT;

defparam L1_ADDER5.input_length = L1_ADDER5_IN_LENGTH;
defparam L1_ADDER5.output_length = L1_ADDER5_OUT_LENGTH;
```

```
bitadder L1_ADDER5
(
	.in
	(
		{
			L0_COMP_OUT[1*GFM_IN_LENGTH+4],
			L0_COMP_OUT[2*GFM_IN_LENGTH+3],
			L0_COMP_OUT[3*GFM_IN_LENGTH+2],
			L0_COMP_OUT[4*GFM_IN_LENGTH+1],
			L0_COMP_OUT[5*GFM_IN_LENGTH+0],
			L0_COMP_OUT[5*GFM_IN_LENGTH+7],
			L0_COMP_OUT[6*GFM_IN_LENGTH+6],
			L0_COMP_OUT[6*GFM_IN_LENGTH+7],
			L0_COMP_OUT[7*GFM_IN_LENGTH+5],
			L0_COMP_OUT[7*GFM_IN_LENGTH+6]

		}
	),
	.out
	(
		{
			L1_ADDER5_OUT
		}
	)
);

//assign LEDR = L1_ADDER5_OUT; // TP LVL1 ADDER5 RESULT

wire [L1_ADDER5_IN_LENGTH/2:0] L1_COMP5;

generate

	genvar L1_COMP5_GEN_I;

	for
	(
		L1_COMP5_GEN_I = 0;
		(L1_COMP5_GEN_I*2+1) <= L1_ADDER5_IN_LENGTH;
		L1_COMP5_GEN_I = L1_COMP5_GEN_I + 1
	)
	begin : L1_COMP5_GENBLOCK
		assign L1_COMP5[ L1_COMP5_GEN_I ] = ( L1_ADDER5_OUT  >=
L1_COMP5_GEN_I*2+1 )?1'b1:1'b0;
	end

endgenerate

//assign LEDR = L1_COMP5; // TP LVL1 COMP5 RESULT

parameter L2_ADDER5_OUT_LENGTH = L1_ADDER5_OUT_LENGTH;

wire [L2_ADDER5_OUT_LENGTH-1:0] L2_ADDER5_OUT;

bitadder L2_ADDER5
(
	.in(L1_COMP5),
	.out(L2_ADDER5_OUT)
);

wire [L2_ADDER5_OUT_LENGTH-1:0] PARITY_SUB5 =  {L2_ADDER5_OUT, 1'b0}-
L1_ADDER5_OUT;
```

```
assign GFM_OUT[5] = ( PARITY_SUB5 >= 1 )?1'b1:1'b0;
```

Saída: C6

```
//a1b5;a2b4;a3b3;a4b2;a5b1;a6b0;a6b7;a7b6;a7b7;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER6_IN_LENGTH = 9;
parameter L1_ADDER6_OUT_LENGTH = 4;

wire [L1_ADDER6_OUT_LENGTH-1:0] L1_ADDER6_OUT;

defparam L1_ADDER6.input_length = L1_ADDER6_IN_LENGTH;
defparam L1_ADDER6.output_length = L1_ADDER6_OUT_LENGTH;

bitadder L1_ADDER6
(
      .in
      (
            {
                  L0_COMP_OUT[1*GFM_IN_LENGTH+5],
                  L0_COMP_OUT[2*GFM_IN_LENGTH+4],
                  L0_COMP_OUT[3*GFM_IN_LENGTH+3],
                  L0_COMP_OUT[4*GFM_IN_LENGTH+2],
                  L0_COMP_OUT[5*GFM_IN_LENGTH+1],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+0],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+7],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+6],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+7]


            }
      ),
      .out
      (
            {
                  L1_ADDER6_OUT
            }
      )
);

//assign LEDR = L1_ADDER6_OUT; // TP LVL1 ADDER6 RESULT

wire [L1_ADDER6_IN_LENGTH/2:0] L1_COMP6;

generate

      genvar L1_COMP6_GEN_I;

      for
      (
            L1_COMP6_GEN_I = 0;
            (L1_COMP6_GEN_I*2+1) <= L1_ADDER6_IN_LENGTH;
            L1_COMP6_GEN_I = L1_COMP6_GEN_I + 1
      )
      begin : L1_COMP6_GENBLOCK
            assign L1_COMP6[ L1_COMP6_GEN_I ] = ( L1_ADDER6_OUT  >=
L1_COMP6_GEN_I*2+1 )?1'b1:1'b0;
      end
```

```
endgenerate

//assign LEDR = L1_COMP6; // TP LVL1 COMP6 RESULT

parameter L2_ADDER6_OUT_LENGTH = L1_ADDER6_OUT_LENGTH;

wire [L2_ADDER6_OUT_LENGTH-1:0] L2_ADDER6_OUT;

bitadder L2_ADDER6
(
      .in(L1_COMP6),
      .out(L2_ADDER6_OUT)
);

wire [L2_ADDER6_OUT_LENGTH-1:0] PARITY_SUB6 =  {L2_ADDER6_OUT, 1'b0}-
L1_ADDER6_OUT;

assign GFM_OUT[6] = ( PARITY_SUB6 >= 1 )?1'b1:1'b0;
```

Saída: C7

```
//a0b7;a1b6;a2b5;a3b4;a4b3;a5b2;a6b1;a7b0;a7b7;
//find: a([0-9])b([0-9]);
//repl: \t\t\tL0_COMP_OUT[\1*GFM_IN_LENGTH+\2],\n

parameter L1_ADDER7_IN_LENGTH = 9;
parameter L1_ADDER7_OUT_LENGTH = 4;

wire [L1_ADDER7_OUT_LENGTH-1:0] L1_ADDER7_OUT;

defparam L1_ADDER7.input_length = L1_ADDER7_IN_LENGTH;
defparam L1_ADDER7.output_length = L1_ADDER7_OUT_LENGTH;

bitadder L1_ADDER7
(
      .in
      (
            {
                  L0_COMP_OUT[0*GFM_IN_LENGTH+7],
                  L0_COMP_OUT[1*GFM_IN_LENGTH+6],
                  L0_COMP_OUT[2*GFM_IN_LENGTH+5],
                  L0_COMP_OUT[3*GFM_IN_LENGTH+4],
                  L0_COMP_OUT[4*GFM_IN_LENGTH+3],
                  L0_COMP_OUT[5*GFM_IN_LENGTH+2],
                  L0_COMP_OUT[6*GFM_IN_LENGTH+1],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+0],
                  L0_COMP_OUT[7*GFM_IN_LENGTH+7]

            }
      ),
      .out
      (
            {
                  L1_ADDER7_OUT
            }
      )
);

//assign LEDR = L1_ADDER7_OUT; // TP LVL1 ADDER7 RESULT
```

```
wire [L1_ADDER7_IN_LENGTH/2:0] L1_COMP7;

generate

      genvar L1_COMP7_GEN_I;

      for
      (
            L1_COMP7_GEN_I = 0;
            (L1_COMP7_GEN_I*2+1) <= L1_ADDER7_IN_LENGTH;
            L1_COMP7_GEN_I = L1_COMP7_GEN_I + 1
      )
      begin : L1_COMP7_GENBLOCK
            assign L1_COMP7[ L1_COMP7_GEN_I ] = ( L1_ADDER7_OUT  >=
L1_COMP7_GEN_I*2+1 )?1'b1:1'b0;
      end

endgenerate

//assign LEDR = L1_COMP7; // TP LVL1 COMP7 RESULT

parameter L2_ADDER7_OUT_LENGTH = L1_ADDER7_OUT_LENGTH;

wire [L2_ADDER7_OUT_LENGTH-1:0] L2_ADDER7_OUT;

bitadder L2_ADDER7
(
      .in(L1_COMP7),
      .out(L2_ADDER7_OUT)
);

wire [L2_ADDER7_OUT_LENGTH-1:0] PARITY_SUB7 =  {L2_ADDER7_OUT, 1'b0}-
L1_ADDER7_OUT;

assign GFM_OUT[7] = ( PARITY_SUB7 >= 1 )?1'b1:1'b0;



module bitadder
(

      input [input_length-1:0] in,
      output [output_length-1:0] out

);

parameter input_length = 8;
parameter output_length = 4; // floor(log2(input_length)+1);

wire [output_length:0] sum [input_length:0];

genvar i;
generate for (i=0;i<input_length;i = i+1) begin : genblock
      assign sum[i+1] = sum[i]+in[i];
end endgenerate

assign out = sum[input_length];
```

```
endmodule
```

TestBench_Paridade

```
#include "VNeuralParityCheck_TestBench.h"
#include "verilated.h"

//vcd traces
#include "verilated_vcd_c.h"

//cout
#include <iostream>

VNeuralParityCheck_TestBench *top; // Instantiation of module

unsigned int main_time = 0; // Current simulation time

double sc_time_stamp () { // Called by $time in Verilog
  return main_time;
}

int main(int argc, char** argv) {
  Verilated::commandArgs(argc, argv);
  top = new VNeuralParityCheck_TestBench;

  while (!Verilated::gotFinish()) {
    top->clk = ~top->clk;
    top->eval(); // Evaluate model
    main_time++; // Time passes...
  } //while

  delete top;
  exit(0);
}


/* verilator lint_off UNSIGNED */
/* verilator lint_off WIDTH */
module Test (

input reg clk

);

parameter bitlength = 13;
parameter deepness = 2**bitlength;

reg [3:0] clk_counter;
reg [bitlength:0] counter;
wire result;

initial begin

    counter = 0;

end
```

```
//always@(posedge clk) begin
//    clk_counter = clk_counter + 1;
//end


always@( posedge clk ) begin

    if ( counter >= deepness )
        $finish;
    else
    begin
        $display("%d %d", counter, result);
        counter = counter + 1;
    end

end
defparam NPC0.IN_LENGTH = bitlength;
NeuralParityCheck NPC0 ( .IN(counter), .OUT(result) );

endmodule
```

TestBench Multiplicador $GF(2^8)$

```
#include "VGF8Multiplier_TestBench.h"
#include "verilated.h"

//vcd traces
#include "verilated_vcd_c.h"

//cout
#include <iostream>

VGF8Multiplier_TestBench *top; // Instantiation of module

unsigned int main_time = 0; // Current simulation time

double sc_time_stamp () { // Called by $time in Verilog
  return main_time;
}

int main(int argc, char** argv) {
  Verilated::commandArgs(argc, argv);
  top = new VGF8Multiplier_TestBench;

  while (!Verilated::gotFinish()) {
    top->clk = ~top->clk;
    top->eval(); // Evaluate model
    main_time++; // Time passes...
  } //while

  delete top;
  exit(0);
}
```

Validação Multiplicador $GF(2^8)$

```
/* verilator lint_off UNSIGNED */
/* verilator lint_off WIDTH */
module Test (

input reg clk

);

reg [31:0] main_counter;
reg [31:0] A;
reg [31:0] B;
wire [31:0] P;

initial begin

     A = 0;
     B = 0;

end


always@(posedge clk) begin


     if ( A < 256 && B < 256 )
          $display("%d %d %d", A, B, P);
     else if ( B == 256 )
          $finish;

     A = A + 1;

     if ( A == 256 )
     begin
          B = B+1;
          A = 0;
     end

end

GF8Multiplier GF8M0 ( .A(A), .B(B), .P(P) );

endmodule
```

Apêndice II

Este apêndice tem por objetivo descrever explicitamente os código em Matlab utilizados para validação e verificação do funcionamento dos módulos de cada paridade e do multiplicador em GF(2^8).

Verificação da Paridade:

```
clear; clc;

!make -j -f NeuralParityCheck_TestBench.mk clean
!make -j -f NeuralParityCheck_TestBench.mk
!make -j -f NeuralParityCheck_TestBench.mk test

NeuralParityCheck_Validator_GenerateReference
NeuralParityCheck_Validator_ImportRawResult

if (any((npc_result_vector-npc_reference_vector) == 0))
    fprintf('Validation: Ok! All results are equal to reference.\n');
else
    fprintf('Validation: Failed!\n');
end

gf8_diff_matrix = gf8_reference_matrix-gf8_result_matrix;
gf8_diff_matrix_offseted = gf8_diff_matrix-min(min(gf8_diff_matrix));
gf8_diff_matrix_scaled =
uint8(255/max(max(gf8_diff_matrix_offseted))*gf8_diff_matrix_offseted)
;
gf8_diff_matrix_spones = full(spones(gf8_diff_matrix));
gf8_diff_matrix_spones_scaled = uint8(255*gf8_diff_matrix_spones);

figure;

sbp1 = subplot(2,2,1);
imshow(uint8(gf8_reference_matrix));
title('GF8 Field Result From Matlab');
colorbar('peer', sbp1);

sbp2 = subplot(2,2,2);
imshow(uint8(gf8_result_matrix));
title('GF8 Field Result From Verilog (Using Verilator)');
colorbar('peer', sbp2);

sbp3 = subplot(2,2,3);
imshow(gf8_diff_matrix_scaled);
title('Difference Between Fields');
colorbar('peer', sbp3);

sbp4 = subplot(2,2,4);
imshow(gf8_diff_matrix_spones_scaled);
title('Elements Difference (Blue: Not Equal, Green: Equal)');
colorbar('peer', sbp4);
colormap('Winter');
```

Verificação do Multiplicador:

```
clear; clc;

!make -j -f GF8Multiplier_TestBench.mk clean
!make -j -f GF8Multiplier_TestBench.mk
!make -j -f GF8Multiplier_TestBench.mk test

GF8Multiplier_Validator_GenerateReference
GF8Multiplier_Validator_ImportRawResult

gf8_diff_matrix = gf8_reference_matrix-gf8_result_matrix;
gf8_diff_matrix_offseted = gf8_diff_matrix-min(min(gf8_diff_matrix));
gf8_diff_matrix_scaled =
uint8(255/max(max(gf8_diff_matrix_offseted))*gf8_diff_matrix_offseted)
;
gf8_diff_matrix_spones = full(spones(gf8_diff_matrix));
gf8_diff_matrix_spones_scaled = uint8(255*gf8_diff_matrix_spones);

figure;

sbp1 = subplot(2,2,1);
imshow(uint8(gf8_reference_matrix));
title('GF8 Field Result From Matlab');
colorbar('peer', sbp1);

sbp2 = subplot(2,2,2);
imshow(uint8(gf8_result_matrix));
title('GF8 Field Result From Verilog (Using Verilator)');
colorbar('peer', sbp2);

sbp3 = subplot(2,2,3);
imshow(gf8_diff_matrix_scaled);
title('Difference Between Fields');
colorbar('peer', sbp3);

sbp4 = subplot(2,2,4);
imshow(gf8_diff_matrix_spones_scaled);
title('Elements Difference (Blue: Not Equal, Green: Equal)');
colorbar('peer', sbp4);
colormap('Winter');
```

Apêndice III

Neste apêndice apresentamos os resultados fornecidos pela ferramenta de descrição, síntese e verificação Quartus® II em termos de número de portas utilizadas, potência, tensão e corrente e tempo mínimo de atraso de propagação necessário para realizar a operação de multiplicação em corpo finito.

PowerPlay Power Analyzer Status	Successful - Mon Sep 21 19:28:27 2015
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Project
Top-level Entity Name	TopLevel
Family	Cyclone II
Device	EP2C35F672C6
Power Models	Final
Total Thermal Power Dissipation	132.58 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	80.85 mW
I/O Thermal Power Dissipation	51.73 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Portas Utilizadas e Fan-Out

Resource	Usage
Estimated Total logic elements	296
Total combinational functions	296
Logic element usage by number of LUT inputs	
-- 4 input functions	188
-- 3 input functions	76
-- <=2 input functions	32
Logic elements by mode	
-- normal mode	259
-- arithmetic mode	37
Total registers	0
-- Dedicated logic registers	0
-- I/O registers	0
I/O pins	182
Embedded Multiplier 9-bit elements	0
Maximum fan-out node	SW[15]
Maximum fan-out	44
Total fan-out	1090
Average fan-out	2.28

Corrente Total

Voltage Supply	Total Current Drawn (1)	Dynamic Current Drawn (1)	Static Current Drawn (1)	Minimum Power Supply Current (2)
VCCINT	79.16 mA	0.00 mA	79.16 mA	79.16 mA
VCCIO	11.39 mA	0.00 mA	11.39 mA	11.39 mA

Corrente em cada pino de entrada e saída

I/O Bank	VCCIO Voltage	Total Current Drawn	Dynamic Current Drawn	Static Current Drawn
1	3.3V	1.44 mA	0.00 mA	1.44 mA
2	3.3V	1.32 mA	0.00 mA	1.32 mA
3	3.3V	1.15 mA	0.00 mA	1.15 mA
4	3.3V	1.15 mA	0.00 mA	1.15 mA
5	3.3V	1.82 mA	0.00 mA	1.82 mA
6	3.3V	1.79 mA	0.00 mA	1.79 mA
7	3.3V	1.44 mA	0.00 mA	1.44 mA
8	3.3V	1.30 mA	0.00 mA	1.30 mA

Corrente Total no conjunto de pinos

VCCIO Voltage	Total Current Drawn (1)	Dynamic Current Drawn (1)	Static Current Drawn (1)	Minimum Power Supply Current (2)
3.3V	11.39 mA	0.00 mA	11.39 mA	11.39 mA

Tempo mínimo de atraso de propagação

Input Port	Output Port	RR	RF	FR	FF
SW[0]	HEX0[0]	5.636	5.636	5.636	5.636
SW[0]	HEX0[1]	5.609	5.609	5.609	5.609
SW[0]	HEX0[2]	5.618	5.618	5.618	5.618
SW[0]	HEX0[3]	5.771	5.771	5.771	5.771
SW[0]	HEX0[4]	5.751	5.751	5.751	5.751
SW[0]	HEX0[5]	5.747	5.747	5.747	5.747
SW[0]	HEX0[6]	5.758	5.758	5.758	5.758
SW[0]	HEX1[0]	7.163	7.163	7.163	7.163
SW[0]	HEX1[1]	7.186	7.186	7.186	7.186
SW[0]	HEX1[2]	6.957	6.957	6.957	6.957
SW[0]	HEX1[3]	6.932	6.932	6.932	6.932
SW[0]	HEX1[4]	6.974	6.974	6.974	6.974
SW[0]	HEX1[5]	7.063	7.063	7.063	7.063
SW[0]	HEX1[6]	7.088	7.088	7.088	7.088
SW[0]	HEX4[0]	3.942	3.942	3.942	3.942
SW[0]	HEX4[1]	3.949	3.949	3.949	3.949
SW[0]	HEX4[2]		3.947	3.947	
SW[0]	HEX4[3]	3.803	3.803	3.803	3.803
SW[0]	HEX4[4]	3.821			3.821
SW[0]	HEX4[5]	3.826			3.826
SW[0]	HEX4[6]	3.945	3.945	3.945	3.945
SW[0]	LEDG[0]	5.410	5.410	5.410	5.410
SW[0]	LEDG[7]	6.651	6.651	6.651	6.651

SW[1]	HEX0[0]	5.376	5.376	5.376	5.376
SW[1]	HEX0[1]	5.353	5.353	5.353	5.353
SW[1]	HEX0[2]	5.364	5.364	5.364	5.364
SW[1]	HEX0[3]	5.512	5.512	5.512	5.512
SW[1]	HEX0[4]	5.495	5.495	5.495	5.495
SW[1]	HEX0[5]	5.492	5.492	5.492	5.492
SW[1]	HEX0[6]	5.502	5.502	5.502	5.502
SW[1]	HEX1[0]	5.816	5.816	5.816	5.816
SW[1]	HEX1[1]	5.836	5.836	5.836	5.836
SW[1]	HEX1[2]	5.608	5.608	5.608	5.608
SW[1]	HEX1[3]	5.583	5.583	5.583	5.583
SW[1]	HEX1[4]	5.617	5.617	5.617	5.617
SW[1]	HEX1[5]	5.708	5.708	5.708	5.708
SW[1]	HEX1[6]	5.739	5.739	5.739	5.739
SW[1]	HEX4[0]	4.073	4.073	4.073	4.073
SW[1]	HEX4[1]	4.082	4.082	4.082	4.082
SW[1]	HEX4[2]	4.079			4.079
SW[1]	HEX4[3]	3.934	3.934	3.934	3.934
SW[1]	HEX4[4]		3.954	3.954	
SW[1]	HEX4[5]	3.950	3.950	3.950	3.950
SW[1]	HEX4[6]	4.078	4.078	4.078	4.078
SW[1]	LEDG[0]	5.575	5.575	5.575	5.575
SW[1]	LEDG[1]	5.613	5.613	5.613	5.613
SW[1]	LEDG[2]	6.659	6.659	6.659	6.659
SW[1]	LEDG[3]	7.785	7.785	7.785	7.785
SW[1]	LEDG[4]	6.542	6.542	6.542	6.542
SW[1]	LEDG[5]	7.309	7.309	7.309	7.309
SW[1]	LEDG[6]	5.329	5.329	5.329	5.329
SW[1]	LEDG[7]	6.807	6.807	6.807	6.807

SW[2]	HEX0[0]	5.831	5.831	5.831	5.831
SW[2]	HEX0[1]	5.808	5.808	5.808	5.808
SW[2]	HEX0[2]	5.819	5.819	5.819	5.819
SW[2]	HEX0[3]	5.967	5.967	5.967	5.967
SW[2]	HEX0[4]	5.950	5.950	5.950	5.950
SW[2]	HEX0[5]	5.947	5.947	5.947	5.947
SW[2]	HEX0[6]	5.957	5.957	5.957	5.957
SW[2]	HEX1[0]	6.400	6.400	6.400	6.400
SW[2]	HEX1[1]	6.420	6.420	6.420	6.420
SW[2]	HEX1[2]	6.192	6.192	6.192	6.192
SW[2]	HEX1[3]	6.167	6.167	6.167	6.167
SW[2]	HEX1[4]	6.201	6.201	6.201	6.201
SW[2]	HEX1[5]	6.292	6.292	6.292	6.292
SW[2]	HEX1[6]	6.323	6.323	6.323	6.323
SW[2]	HEX4[0]	3.924	3.924	3.924	3.924
SW[2]	HEX4[1]	3.942			3.942
SW[2]	HEX4[2]	3.930	3.930	3.930	3.930
SW[2]	HEX4[3]	3.795	3.795	3.795	3.795
SW[2]	HEX4[4]	3.814	3.814	3.814	3.814
SW[2]	HEX4[5]	3.811	3.811	3.811	3.811
SW[2]	HEX4[6]	3.938	3.938	3.938	3.938
SW[2]	LEDG[0]	6.225	6.225	6.225	6.225
SW[2]	LEDG[1]	6.068	6.068	6.068	6.068
SW[2]	LEDG[2]	7.159	7.159	7.159	7.159
SW[2]	LEDG[3]	7.994	7.994	7.994	7.994
SW[2]	LEDG[4]	6.803	6.803	6.803	6.803
SW[2]	LEDG[5]	7.160	7.160	7.160	7.160
SW[2]	LEDG[6]	5.913	5.913	5.913	5.913
SW[2]	LEDG[7]	7.071	7.071	7.071	7.071

SW[3]	HEX0[0]	6.105	6.105	6.105	6.105
SW[3]	HEX0[1]	6.078	6.078	6.078	6.078
SW[3]	HEX0[2]	6.087	6.087	6.087	6.087
SW[3]	HEX0[3]	6.240	6.240	6.240	6.240
SW[3]	HEX0[4]	6.220	6.220	6.220	6.220
SW[3]	HEX0[5]	6.216	6.216	6.216	6.216
SW[3]	HEX0[6]	6.227	6.227	6.227	6.227
SW[3]	HEX1[0]	6.055	6.055	6.055	6.055
SW[3]	HEX1[1]	6.075	6.075	6.075	6.075
SW[3]	HEX1[2]	5.847	5.847	5.847	5.847
SW[3]	HEX1[3]	5.822	5.822	5.822	5.822
SW[3]	HEX1[4]	5.856	5.856	5.856	5.856
SW[3]	HEX1[5]	5.947	5.947	5.947	5.947
SW[3]	HEX1[6]	5.978	5.978	5.978	5.978
SW[3]	HEX4[0]	3.581	3.581	3.581	3.581
SW[3]	HEX4[1]	3.598	3.598	3.598	3.598
SW[3]	HEX4[2]	3.590	3.590	3.590	3.590
SW[3]	HEX4[3]	3.451	3.451	3.451	3.451
SW[3]	HEX4[4]		3.471	3.471	
SW[3]	HEX4[5]	3.467	3.467	3.467	3.467
SW[3]	HEX4[6]	3.594	3.594	3.594	3.594
SW[3]	LEDG[0]	5.879	5.879	5.879	5.879
SW[3]	LEDG[1]	6.445	6.445	6.445	6.445
SW[3]	LEDG[2]	6.977	6.977	6.977	6.977
SW[3]	LEDG[3]	7.253	7.253	7.253	7.253
SW[3]	LEDG[4]	6.460	6.460	6.460	6.460
SW[3]	LEDG[5]	7.305	7.305	7.305	7.305
SW[3]	LEDG[6]	5.568	5.568	5.568	5.568
SW[3]	LEDG[7]	7.150	7.150	7.150	7.150

SW[4]	HEX0[0]	6.255	6.255	6.255	6.255
SW[4]	HEX0[1]	6.228	6.228	6.228	6.228
SW[4]	HEX0[2]	6.237	6.237	6.237	6.237
SW[4]	HEX0[3]	6.390	6.390	6.390	6.390
SW[4]	HEX0[4]	6.370	6.370	6.370	6.370
SW[4]	HEX0[5]	6.366	6.366	6.366	6.366
SW[4]	HEX0[6]	6.377	6.377	6.377	6.377
SW[4]	HEX1[0]	6.464	6.464	6.464	6.464
SW[4]	HEX1[1]	6.484	6.484	6.484	6.484
SW[4]	HEX1[2]	6.256	6.256	6.256	6.256
SW[4]	HEX1[3]	6.231	6.231	6.231	6.231
SW[4]	HEX1[4]	6.265	6.265	6.265	6.265
SW[4]	HEX1[5]	6.356	6.356	6.356	6.356
SW[4]	HEX1[6]	6.387	6.387	6.387	6.387
SW[4]	HEX5[0]	3.631	3.631	3.631	3.631
SW[4]	HEX5[1]	3.592	3.592	3.592	3.592
SW[4]	HEX5[2]		3.603	3.603	
SW[4]	HEX5[3]	3.705	3.705	3.705	3.705
SW[4]	HEX5[4]	3.736			3.736
SW[4]	HEX5[5]	3.741			3.741
SW[4]	HEX5[6]	3.757	3.757	3.757	3.757
SW[4]	LEDG[0]	6.029	6.029	6.029	6.029
SW[4]	LEDG[1]	6.566	6.566	6.566	6.566
SW[4]	LEDG[2]	7.316	7.316	7.316	7.316
SW[4]	LEDG[3]	7.614	7.614	7.614	7.614
SW[4]	LEDG[4]	6.770	6.770	6.770	6.770
SW[4]	LEDG[5]	7.171	7.171	7.171	7.171
SW[4]	LEDG[6]	5.977	5.977	5.977	5.977
SW[4]	LEDG[7]	7.290	7.290	7.290	7.290

SW[5]	HEX0[0]	5.804	5.804	5.804	5.804
SW[5]	HEX0[1]	5.777	5.777	5.777	5.777
SW[5]	HEX0[2]	5.786	5.786	5.786	5.786
SW[5]	HEX0[3]	5.939	5.939	5.939	5.939
SW[5]	HEX0[4]	5.919	5.919	5.919	5.919
SW[5]	HEX0[5]	5.915	5.915	5.915	5.915
SW[5]	HEX0[6]	5.926	5.926	5.926	5.926
SW[5]	HEX1[0]	6.573	6.573	6.573	6.573
SW[5]	HEX1[1]	6.593	6.593	6.593	6.593
SW[5]	HEX1[2]	6.365	6.365	6.365	6.365
SW[5]	HEX1[3]	6.340	6.340	6.340	6.340
SW[5]	HEX1[4]	6.374	6.374	6.374	6.374
SW[5]	HEX1[5]	6.465	6.465	6.465	6.465
SW[5]	HEX1[6]	6.496	6.496	6.496	6.496
SW[5]	HEX5[0]	3.619	3.619	3.619	3.619
SW[5]	HEX5[1]	3.577	3.577	3.577	3.577
SW[5]	HEX5[2]	3.590			3.590
SW[5]	HEX5[3]	3.693	3.693	3.693	3.693
SW[5]	HEX5[4]		3.721	3.721	
SW[5]	HEX5[5]	3.722	3.722	3.722	3.722
SW[5]	HEX5[6]	3.743	3.743	3.743	3.743
SW[5]	LEDG[0]	5.578	5.578	5.578	5.578
SW[5]	LEDG[1]	6.089	6.089	6.089	6.089
SW[5]	LEDG[2]	7.742	7.742	7.742	7.742
SW[5]	LEDG[3]	7.545	7.545	7.545	7.545
SW[5]	LEDG[4]	7.069	7.069	7.069	7.069
SW[5]	LEDG[5]	7.145	7.145	7.145	7.145
SW[5]	LEDG[6]	6.086	6.086	6.086	6.086
SW[5]	LEDG[7]	7.002	7.002	7.002	7.002

SW[6]	HEX0[0]	6.016	6.016	6.016	6.016
SW[6]	HEX0[1]	5.993	5.993	5.993	5.993
SW[6]	HEX0[2]	6.004	6.004	6.004	6.004
SW[6]	HEX0[3]	6.152	6.152	6.152	6.152
SW[6]	HEX0[4]	6.135	6.135	6.135	6.135
SW[6]	HEX0[5]	6.132	6.132	6.132	6.132
SW[6]	HEX0[6]	6.142	6.142	6.142	6.142
SW[6]	HEX1[0]	6.109	6.109	6.109	6.109
SW[6]	HEX1[1]	6.129	6.129	6.129	6.129
SW[6]	HEX1[2]	5.901	5.901	5.901	5.901
SW[6]	HEX1[3]	5.876	5.876	5.876	5.876
SW[6]	HEX1[4]	5.910	5.910	5.910	5.910
SW[6]	HEX1[5]	6.001	6.001	6.001	6.001
SW[6]	HEX1[6]	6.032	6.032	6.032	6.032
SW[6]	HEX5[0]	3.526	3.526	3.526	3.526
SW[6]	HEX5[1]	3.480			3.480
SW[6]	HEX5[2]	3.496	3.496	3.496	3.496
SW[6]	HEX5[3]	3.602	3.602	3.602	3.602
SW[6]	HEX5[4]	3.626	3.626	3.626	3.626
SW[6]	HEX5[5]	3.629	3.629	3.629	3.629
SW[6]	HEX5[6]	3.649	3.649	3.649	3.649
SW[6]	LEDG[0]	6.036	6.036	6.036	6.036
SW[6]	LEDG[1]	6.253	6.253	6.253	6.253
SW[6]	LEDG[2]	7.888	7.888	7.888	7.888
SW[6]	LEDG[3]	8.046	8.046	8.046	8.046
SW[6]	LEDG[4]	7.016	7.016	7.016	7.016
SW[6]	LEDG[5]	6.977	6.977	6.977	6.977
SW[6]	LEDG[6]	5.622	5.622	5.622	5.622
SW[6]	LEDG[7]	7.154	7.154	7.154	7.154

SW[7]	HEX0[0]	6.165	6.165	6.165	6.165
SW[7]	HEX0[1]	6.138	6.138	6.138	6.138
SW[7]	HEX0[2]	6.147	6.147	6.147	6.147
SW[7]	HEX0[3]	6.300	6.300	6.300	6.300
SW[7]	HEX0[4]	6.280	6.280	6.280	6.280
SW[7]	HEX0[5]	6.276	6.276	6.276	6.276
SW[7]	HEX0[6]	6.287	6.287	6.287	6.287
SW[7]	HEX1[0]	6.227	6.227	6.227	6.227
SW[7]	HEX1[1]	6.247	6.247	6.247	6.247
SW[7]	HEX1[2]	6.019	6.019	6.019	6.019
SW[7]	HEX1[3]	5.994	5.994	5.994	5.994
SW[7]	HEX1[4]	6.028	6.028	6.028	6.028
SW[7]	HEX1[5]	6.119	6.119	6.119	6.119
SW[7]	HEX1[6]	6.150	6.150	6.150	6.150
SW[7]	HEX5[0]	3.593	3.593	3.593	3.593
SW[7]	HEX5[1]	3.562	3.562	3.562	3.562
SW[7]	HEX5[2]	3.568	3.568	3.568	3.568
SW[7]	HEX5[3]	3.669	3.669	3.669	3.669
SW[7]	HEX5[4]		3.700	3.700	
SW[7]	HEX5[5]	3.698	3.698	3.698	3.698
SW[7]	HEX5[6]	3.727	3.727	3.727	3.727
SW[7]	LEDG[0]	5.939	5.939	5.939	5.939
SW[7]	LEDG[1]	6.466	6.466	6.466	6.466
SW[7]	LEDG[2]	7.644	7.644	7.644	7.644
SW[7]	LEDG[3]	7.492	7.492	7.492	7.492
SW[7]	LEDG[4]	6.965	6.965	6.965	6.965
SW[7]	LEDG[5]	7.018	7.018	7.018	7.018
SW[7]	LEDG[6]	5.740	5.740	5.740	5.740
SW[7]	LEDG[7]	6.983	6.983	6.983	6.983

SW[8]	HEX0[0]	5.575	5.575	5.575	5.575
SW[8]	HEX0[1]	5.552	5.552	5.552	5.552
SW[8]	HEX0[2]	5.563	5.563	5.563	5.563
SW[8]	HEX0[3]	5.711	5.711	5.711	5.711
SW[8]	HEX0[4]	5.694	5.694	5.694	5.694
SW[8]	HEX0[5]	5.691	5.691	5.691	5.691
SW[8]	HEX0[6]	5.701	5.701	5.701	5.701
SW[8]	HEX1[0]	6.039	6.039	6.039	6.039
SW[8]	HEX1[1]	6.059	6.059	6.059	6.059
SW[8]	HEX1[2]	5.831	5.831	5.831	5.831
SW[8]	HEX1[3]	5.806	5.806	5.806	5.806
SW[8]	HEX1[4]	5.840	5.840	5.840	5.840
SW[8]	HEX1[5]	5.931	5.931	5.931	5.931
SW[8]	HEX1[6]	5.962	5.962	5.962	5.962
SW[8]	HEX6[0]	3.800	3.800	3.800	3.800
SW[8]	HEX6[1]	3.931	3.931	3.931	3.931
SW[8]	HEX6[2]		3.849	3.849	
SW[8]	HEX6[3]	4.038	4.038	4.038	4.038
SW[8]	HEX6[4]	4.038			4.038
SW[8]	HEX6[5]	4.022			4.022
SW[8]	HEX6[6]	4.012	4.012	4.012	4.012
SW[8]	LEDG[0]	5.375	5.375	5.375	5.375
SW[8]	LEDG[1]	5.812	5.812	5.812	5.812
SW[8]	LEDG[2]	6.428	6.428	6.428	6.428
SW[8]	LEDG[3]	7.576	7.576	7.576	7.576
SW[8]	LEDG[4]	6.925	6.925	6.925	6.925
SW[8]	LEDG[5]	6.870	6.870	6.870	6.870
SW[8]	LEDG[6]	5.552	5.552	5.552	5.552
SW[8]	LEDG[7]	7.243	7.243	7.243	7.243

SW[9]	HEX0[0]	6.098	6.098	6.098	6.098
SW[9]	HEX0[1]	6.068	6.068	6.068	6.068
SW[9]	HEX0[2]	6.069	6.069	6.069	6.069
SW[9]	HEX0[3]	6.233	6.233	6.233	6.233
SW[9]	HEX0[4]	6.211	6.211	6.211	6.211
SW[9]	HEX0[5]	6.200	6.200	6.200	6.200
SW[9]	HEX0[6]	6.217	6.217	6.217	6.217
SW[9]	HEX1[0]	6.772	6.772	6.772	6.772
SW[9]	HEX1[1]	6.792	6.792	6.792	6.792
SW[9]	HEX1[2]	6.564	6.564	6.564	6.564
SW[9]	HEX1[3]	6.539	6.539	6.539	6.539
SW[9]	HEX1[4]	6.573	6.573	6.573	6.573
SW[9]	HEX1[5]	6.664	6.664	6.664	6.664
SW[9]	HEX1[6]	6.695	6.695	6.695	6.695
SW[9]	HEX6[0]	3.636	3.636	3.636	3.636
SW[9]	HEX6[1]	3.770	3.770	3.770	3.770
SW[9]	HEX6[2]	3.687			3.687
SW[9]	HEX6[3]	3.877	3.877	3.877	3.877
SW[9]	HEX6[4]		3.865	3.865	
SW[9]	HEX6[5]	3.857	3.857	3.857	3.857
SW[9]	HEX6[6]	3.849	3.849	3.849	3.849
SW[9]	LEDG[1]	6.664	6.664	6.664	6.664
SW[9]	LEDG[2]	6.818	6.818	6.818	6.818
SW[9]	LEDG[3]	7.870	7.870	7.870	7.870
SW[9]	LEDG[4]	6.610	6.610	6.610	6.610
SW[9]	LEDG[5]	7.483	7.483	7.483	7.483
SW[9]	LEDG[6]	6.285	6.285	6.285	6.285
SW[9]	LEDG[7]	7.479	7.479	7.479	7.479

SW[10]	HEX0[0]	5.508	5.508	5.508	5.508
SW[10]	HEX0[1]	5.485	5.485	5.485	5.485
SW[10]	HEX0[2]	5.496	5.496	5.496	5.496
SW[10]	HEX0[3]	5.644	5.644	5.644	5.644
SW[10]	HEX0[4]	5.627	5.627	5.627	5.627
SW[10]	HEX0[5]	5.624	5.624	5.624	5.624
SW[10]	HEX0[6]	5.634	5.634	5.634	5.634
SW[10]	HEX1[0]	5.863	5.863	5.863	5.863
SW[10]	HEX1[1]	5.883	5.883	5.883	5.883
SW[10]	HEX1[2]	5.655	5.655	5.655	5.655
SW[10]	HEX1[3]	5.630	5.630	5.630	5.630
SW[10]	HEX1[4]	5.664	5.664	5.664	5.664
SW[10]	HEX1[5]	5.755	5.755	5.755	5.755
SW[10]	HEX1[6]	5.786	5.786	5.786	5.786
SW[10]	HEX6[0]	2.640	2.640	2.640	2.640
SW[10]	HEX6[1]	2.774			2.774
SW[10]	HEX6[2]	2.692	2.692	2.692	2.692
SW[10]	HEX6[3]	2.881	2.881	2.881	2.881
SW[10]	HEX6[4]	2.875	2.875	2.875	2.875
SW[10]	HEX6[5]	2.862	2.862	2.862	2.862
SW[10]	HEX6[6]	2.852	2.852	2.852	2.852
SW[10]	LEDG[1]	5.745	5.745	5.745	5.745
SW[10]	LEDG[2]	7.241	7.241	7.241	7.241
SW[10]	LEDG[3]	6.811	6.811	6.811	6.811
SW[10]	LEDG[4]	5.724	5.724	5.724	5.724
SW[10]	LEDG[5]	6.568	6.568	6.568	6.568
SW[10]	LEDG[6]	5.376	5.376	5.376	5.376
SW[10]	LEDG[7]	6.253	6.253	6.253	6.253

SW[11]	HEX0[0]	5.139	5.139	5.139	5.139
SW[11]	HEX0[1]	5.116	5.116	5.116	5.116
SW[11]	HEX0[2]	5.127	5.127	5.127	5.127
SW[11]	HEX0[3]	5.275	5.275	5.275	5.275
SW[11]	HEX0[4]	5.258	5.258	5.258	5.258
SW[11]	HEX0[5]	5.255	5.255	5.255	5.255
SW[11]	HEX0[6]	5.265	5.265	5.265	5.265
SW[11]	HEX1[0]	5.409	5.409	5.409	5.409
SW[11]	HEX1[1]	5.429	5.429	5.429	5.429
SW[11]	HEX1[2]	5.201	5.201	5.201	5.201
SW[11]	HEX1[3]	5.176	5.176	5.176	5.176
SW[11]	HEX1[4]	5.210	5.210	5.210	5.210
SW[11]	HEX1[5]	5.301	5.301	5.301	5.301
SW[11]	HEX1[6]	5.332	5.332	5.332	5.332
SW[11]	HEX6[0]	2.642	2.642	2.642	2.642
SW[11]	HEX6[1]	2.774	2.774	2.774	2.774
SW[11]	HEX6[2]	2.689	2.689	2.689	2.689
SW[11]	HEX6[3]	2.884	2.884	2.884	2.884
SW[11]	HEX6[4]		2.883	2.883	
SW[11]	HEX6[5]	2.870	2.870	2.870	2.870
SW[11]	HEX6[6]	2.860	2.860	2.860	2.860
SW[11]	LEDG[1]	5.376	5.376	5.376	5.376
SW[11]	LEDG[2]	7.174	7.174	7.174	7.174
SW[11]	LEDG[3]	7.024	7.024	7.024	7.024
SW[11]	LEDG[4]	5.576	5.576	5.576	5.576
SW[11]	LEDG[5]	6.093	6.093	6.093	6.093
SW[11]	LEDG[6]	4.922	4.922	4.922	4.922
SW[11]	LEDG[7]	6.492	6.492	6.492	6.492

SW[12]	HEX0[0]	5.602	5.602	5.602	5.602
SW[12]	HEX0[1]	5.579	5.579	5.579	5.579
SW[12]	HEX0[2]	5.590	5.590	5.590	5.590
SW[12]	HEX0[3]	5.738	5.738	5.738	5.738
SW[12]	HEX0[4]	5.721	5.721	5.721	5.721
SW[12]	HEX0[5]	5.718	5.718	5.718	5.718
SW[12]	HEX0[6]	5.728	5.728	5.728	5.728
SW[12]	HEX1[0]	5.376	5.376	5.376	5.376
SW[12]	HEX1[1]	5.396	5.396	5.396	5.396
SW[12]	HEX1[2]	5.168	5.168	5.168	5.168
SW[12]	HEX1[3]	5.143	5.143	5.143	5.143
SW[12]	HEX1[4]	5.177	5.177	5.177	5.177
SW[12]	HEX1[5]	5.268	5.268	5.268	5.268
SW[12]	HEX1[6]	5.299	5.299	5.299	5.299
SW[12]	HEX7[0]	3.027	3.027	3.027	3.027
SW[12]	HEX7[1]	3.020	3.020	3.020	3.020
SW[12]	HEX7[2]		2.981	2.981	
SW[12]	HEX7[3]	2.982	2.982	2.982	2.982
SW[12]	HEX7[4]	2.988			2.988
SW[12]	HEX7[5]	3.016			3.016
SW[12]	HEX7[6]	3.018	3.018	3.018	3.018
SW[12]	LEDG[1]	5.839	5.839	5.839	5.839
SW[12]	LEDG[2]	7.232	7.232	7.232	7.232
SW[12]	LEDG[3]	7.112	7.112	7.112	7.112
SW[12]	LEDG[4]	6.301	6.301	6.301	6.301
SW[12]	LEDG[5]	6.133	6.133	6.133	6.133
SW[12]	LEDG[6]	4.889	4.889	4.889	4.889
SW[12]	LEDG[7]	6.530	6.530	6.530	6.530

SW[13]	HEX0[0]	8.066	8.066	8.066	8.066
SW[13]	HEX0[1]	8.043	8.043	8.043	8.043
SW[13]	HEX0[2]	8.054	8.054	8.054	8.054
SW[13]	HEX0[3]	8.202	8.202	8.202	8.202
SW[13]	HEX0[4]	8.185	8.185	8.185	8.185
SW[13]	HEX0[5]	8.182	8.182	8.182	8.182
SW[13]	HEX0[6]	8.192	8.192	8.192	8.192
SW[13]	HEX1[0]	7.410	7.410	7.410	7.410
SW[13]	HEX1[1]	7.430	7.430	7.430	7.430
SW[13]	HEX1[2]	7.202	7.202	7.202	7.202
SW[13]	HEX1[3]	7.177	7.177	7.177	7.177
SW[13]	HEX1[4]	7.211	7.211	7.211	7.211
SW[13]	HEX1[5]	7.302	7.302	7.302	7.302
SW[13]	HEX1[6]	7.333	7.333	7.333	7.333
SW[13]	HEX7[0]	5.780	5.780	5.780	5.780
SW[13]	HEX7[1]	5.778	5.778	5.778	5.778
SW[13]	HEX7[2]	5.738			5.738
SW[13]	HEX7[3]	5.743	5.743	5.743	5.743
SW[13]	HEX7[4]		5.738	5.738	
SW[13]	HEX7[5]	5.773	5.773	5.773	5.773
SW[13]	HEX7[6]	5.779	5.779	5.779	5.779
SW[13]	LEDG[1]	8.303	8.303	8.303	8.303
SW[13]	LEDG[2]	9.154	9.154	9.154	9.154
SW[13]	LEDG[3]	9.609	9.609	9.609	9.609
SW[13]	LEDG[4]	8.767	8.767	8.767	8.767
SW[13]	LEDG[5]	8.830	8.830	8.830	8.830
SW[13]	LEDG[6]	6.923	6.923	6.923	6.923
SW[13]	LEDG[7]	8.664	8.664	8.664	8.664

SW[14]	HEX0[0]	7.324	7.324	7.324	7.324
SW[14]	HEX0[1]	7.301	7.301	7.301	7.301
SW[14]	HEX0[2]	7.312	7.312	7.312	7.312
SW[14]	HEX0[3]	7.460	7.460	7.460	7.460
SW[14]	HEX0[4]	7.443	7.443	7.443	7.443
SW[14]	HEX0[5]	7.440	7.440	7.440	7.440
SW[14]	HEX0[6]	7.450	7.450	7.450	7.450
SW[14]	HEX1[0]	7.800	7.800	7.800	7.800
SW[14]	HEX1[1]	7.820	7.820	7.820	7.820
SW[14]	HEX1[2]	7.592	7.592	7.592	7.592
SW[14]	HEX1[3]	7.567	7.567	7.567	7.567
SW[14]	HEX1[4]	7.601	7.601	7.601	7.601
SW[14]	HEX1[5]	7.692	7.692	7.692	7.692
SW[14]	HEX1[6]	7.723	7.723	7.723	7.723
SW[14]	HEX7[0]	5.694	5.694	5.694	5.694
SW[14]	HEX7[1]	5.693			5.693
SW[14]	HEX7[2]	5.647	5.647	5.647	5.647
SW[14]	HEX7[3]	5.653	5.653	5.653	5.653
SW[14]	HEX7[4]	5.651	5.651	5.651	5.651
SW[14]	HEX7[5]	5.682	5.682	5.682	5.682
SW[14]	HEX7[6]	5.689	5.689	5.689	5.689
SW[14]	LEDG[1]	7.561	7.561	7.561	7.561
SW[14]	LEDG[2]	9.125	9.125	9.125	9.125
SW[14]	LEDG[3]	9.278	9.278	9.278	9.278
SW[14]	LEDG[4]	8.692	8.692	8.692	8.692
SW[14]	LEDG[5]	9.105	9.105	9.105	9.105
SW[14]	LEDG[6]	7.313	7.313	7.313	7.313
SW[14]	LEDG[7]	8.445	8.445	8.445	8.445

SW[15]	HEX0[0]	7.576	7.576	7.576	7.576
SW[15]	HEX0[1]	7.553	7.553	7.553	7.553
SW[15]	HEX0[2]	7.564	7.564	7.564	7.564
SW[15]	HEX0[3]	7.712	7.712	7.712	7.712
SW[15]	HEX0[4]	7.695	7.695	7.695	7.695
SW[15]	HEX0[5]	7.692	7.692	7.692	7.692
SW[15]	HEX0[6]	7.702	7.702	7.702	7.702
SW[15]	HEX1[0]	7.798	7.798	7.798	7.798
SW[15]	HEX1[1]	7.818	7.818	7.818	7.818
SW[15]	HEX1[2]	7.590	7.590	7.590	7.590
SW[15]	HEX1[3]	7.565	7.565	7.565	7.565
SW[15]	HEX1[4]	7.599	7.599	7.599	7.599
SW[15]	HEX1[5]	7.690	7.690	7.690	7.690
SW[15]	HEX1[6]	7.721	7.721	7.721	7.721
SW[15]	HEX7[0]	5.871	5.871	5.871	5.871
SW[15]	HEX7[1]	5.871	5.871	5.871	5.871
SW[15]	HEX7[2]	5.830	5.830	5.830	5.830
SW[15]	HEX7[3]	5.829	5.829	5.829	5.829
SW[15]	HEX7[4]		5.829	5.829	
SW[15]	HEX7[5]	5.866	5.866	5.866	5.866
SW[15]	HEX7[6]	5.864	5.864	5.864	5.864
SW[15]	LEDG[0]	7.434	7.434	7.434	7.434
SW[15]	LEDG[1]	7.813	7.813	7.813	7.813
SW[15]	LEDG[2]	8.391	8.391	8.391	8.391
SW[15]	LEDG[3]	9.175	9.175	9.175	9.175
SW[15]	LEDG[4]	8.700	8.700	8.700	8.700
SW[15]	LEDG[5]	8.762	8.762	8.762	8.762
SW[15]	LEDG[6]	7.311	7.311	7.311	7.311
SW[15]	LEDG[7]	8.355	8.355	8.355	8.355

Apêndice IV

Neste apêndice apresentamos os diferentes níveis de abstrações obtidos do RTL *Viewer* da Figura 5.8 e da descrição de *hardware* em Verilog do Apêndice I.