Diogo Anache de Souza

Conjunto Universal de Circuitos Lógicos Quaternários Reversíveis

Campo Grande (MS) - Brasil
Dezembro de 2021

Diogo Anache de Souza

Conjunto Universal de Circuitos Lógicos Quaternários Reversíveis

Dissertação apresentada ao Programa de Mestrado Stricto Sensu em Computação Aplicada, mantido pela Universidade Federal do Mato Grosso do Sul, como parte dos requisitos para a obtenção do título de Mestre em Computação Aplicada.

Orientador:	Prof. Dr. Milton Ernesto Romero Romero
Coorientador:	Prof. Dr. Evandro Mazina Martins
Banca:	Prof. Dr. Fábio Iaione
Banca:	Prof. Dr. Luciano Gonda

Campo Grande (MS) - Brasil
Dezembro de 2021

Prefácio

O tópico III do Art. 34 do Regulamento do Curso de Mestrado Profissional em Computação Aplicada prevê a necessidade de artigo aceito em periódico ou conferência com Qualis restrito (A1, A2, A3 ou A4), com comprovação do aceite e da classificação no Qualis, como comprovação para a etapa de defesa do mestrado. Tal artigo, de título Universal Set of Reversible Quaternary Logic Gates, foi aceito e publicado no periódico International Journal of Computer Applications, de Qualis A4, sendo a maior contribuição do mestrado. Contudo, dada a densidade da pesquisa, optou-se pela produção deste documento, cujo objetivo é fornecer uma base de conhecimentos necessários para compreender o artigo em sua totalidade, além de cobrir um tópico intimamente relacionado com o tema do artigo e que não foi coberto por este, que é a minimização de circuitos lógicos, objeto de estudo do Trabalho de Conclusão de Curso da graduação em Engenharia de Computação, e também durante todo o período como mestrando. A área de concentração na qual o trabalho se encontra é Tecnologias Computacionais para Cidades Inteligentes, na linha de pesquisa Sistemas Computacionais Aplicados à Infraestrutura.

Resumo

A síntese de circuitos digitais em lógica binária utiliza técnicas de minimização, como os mapas de Karnaugh e os algoritmos Quine-McCluskey, Petrick e Espresso, para obter uma expressão equivalente, porém com menos termos e operações, o que implica em um uso reduzido de portas lógicas. É possível utilizar a lógica de múltiplos valores (MVL) para transmitir mais informação por interconexão. Seguindo a ideia de otimização, pode-se também reduzir a dissipação de energia desses circuitos através de portas lógicas reversíveis, que permitem um mapeamento bijetivo entre entrada e saída. Tal conceito apoia-se no princípio de Landauer, que enuncia que a cada bit perdido de informação, K*T*ln2 Joules de energia são dissipados. Neste trabalho foi abordada a álgebra quaternária, sendo proposta uma metodologia de minimização para esse domínio, bem como o projeto de portas lógicas reversíveis.

Palavras-chave: Lógica de múltiplos valores, álgebra quaternária, minimização de circuitos lógicos, portas lógicas reversíveis.

1 Introdução

Em sua forma mais usual, circuitos digitais combinacionais são sintetizados em lógica binária (valores 0 e 1). Para realizar essa tarefa, tem-se a utilização de tabelas verdade para simplificar as expressões lógicas por meio das propriedades, postulados, e equivalências da álgebra Booleana [1]. Dessa técnica foram derivadas algumas ferramentas que facilitam essa simplificação, como o mapa de Karnaugh e os algoritmos de Quine-McCluskey, de Petrick e Espresso. Contudo, a utilização dessas técnicas para dois valores lógicos, embora bem fundamentadas e funcionais, pode apresentar certas limitações devido ao grau de complexidade de um sistema, como o tamanho do chip e o consumo de energia devido ao elevado número de interconexões no circuito [2]. Diversos trabalhos tratam especificamente sobre esse tema [3] [4], analisando a complexidade computacional do problema da minimização de circuitos lógicos, argumentando que pode se encaixar na categoria NP-completo, que são os problemas que podem ser verificados (mas não resolvidos) em tempo polinomial, e se cuja solução for encontrada em tempo polinomial, então todos os outros problemas NP também terão solução em tempo polinomial.

Tais problemas podem ser solucionados a partir do uso conjunto da lógica de múltiplos valores, cuja origem se deu a partir das pesquisas de Łukasiewicz [5], Post [6] e Kleene [7], e da computação reversível [8] [9]. Este trabalho tratará especificamente da álgebra quaternária vista em [10], que baseia-se em conjuntos universais de portas lógicas, onde os operadores Sucessor (SUC), Máximo (MAX) e Produto Estendido ($eAND_i$) representam a forma canônica SOEP (soma de produtos estendidos), e também o seu dual, a forma canônica POES (produto de somas estendidas), que envolve o mesmo Sucessor e os operadores Mínimo (MIN) e Soma Estendida (eOR_i), para i podendo assumir quaisquer valores do domínio quaternário (0, 1, 2, 3). A álgebra proposta tem como vantagem utilizar o conhecimento já existente acerca da síntese em circuitos binários, apenas ampliando-o para o domínio quaternário a partir de modificações pontuais na construção das portas lógicas. Além de diminuir o número de interconexões, devido ao fato de enviar mais informações por conexão, a lógica de múltiplos valores também diminui a área do chip, visto que estas representam cerca de 70% de sua área total [11] [12].

Já a computação reversível tem como base o princípio de Landauer [13], que denota que a cada bit perdido de informação, que é o que acontece em cada porta lógica AND e OR nos circuitos combinacionais, K*T*ln2 Joules de energia são dissipados, onde K é a constante de Boltzmann, T é a temperatura em graus Kelvin, ln é o logaritmo neperiano e o número 2 origina-se da

base binária. Portanto, ao se criar um mapeamento bijetivo entre entrada e saída, diminui-se a dissipação de energia [14] [15] [16]. A implementação das portas lógicas reversíveis consiste em três subsistemas: o primeiro subsistema discrimina cada nível lógico, o segundo subsistema executa a lógica referente à porta, e o terceiro subsistema define o nível lógico de saída.

2 Conceitos Básicos

O mapa de Karnaugh é uma ferramenta de minimização de circuitos lógicos, criado por Maurice Karnaugh [17]. Todavia, por ser um método gráfico, apresenta limitações a partir de seis variáveis, uma vez que depende da habilidade humana de reconhecer padrões, cujo grau de confiabilidade decai conforme o número de variáveis analisadas cresce [18]. Tal entrave pode ser contornado a partir de algoritmos que se aproveitam da organização que o método promove para gerar simplificações.

2.1 Mapa de Karnaugh Quaternário

Define-se uma variável MVL como um literal que pode assumir qualquer valor pertencente ao conjunto ordenado D = {0, 1, ..., L}, onde D é o domínio e L é o elemento superior de D e igual a N - 1, para N sendo o valor da representação (2 para binário, 4 para quaternário) [11]. Tomando o domínio quaternário, apresentam-se as tabelas verdade para os operadores dessa álgebra, onde a Tabela 1 refere-se ao operador Sucessor, a Tabela 2 refere-se ao operador Máximo, a Tabela 3 refere-se ao operador Produto Estendido, a Tabela 4 refere-se ao operador Mínimo e a Tabela 5 refere-se ao operador Soma Estendida. A precedência entre as operações se dá da seguinte forma: $SUC > eAND_i = eOR_i > MAX = MIN$.

Tabela 1: Tabela verdade para o operador Sucessor (A^i).

A	A^1	A^2	A^3
0	1	2	3
1	2	3	0
2	3	0	1
3	0	1	2

Tabela 2: Tabela verdade para o operador Máximo (+).

A\B	0	1	2	3
0	0	1	2	3
1	1	1	2	3
2	2	2	2	3
3	3	3	3	3

Tabela 3: Tabela verdade para o operador Produto Estendido ($*^1$, $*^2$, $*^3$).

A\B	0	1	2	3
0	0	0	0	0
1	0	1	0	0
2	0	0	0	0
3	0	0	0	0

A\B	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	2	0
3	0	0	0	0

A\B	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	3

Tabela 4: Tabela verdade para o operador Mínimo ($\cdot$).

A\B	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	2	2
3	0	1	2	3

Tabela 5: Tabela verdade para o operador Soma Estendida ($+^1$, $+^2$, $+^3$).

A\B	0	1	2	3
0	0	3	3	3
1	3	3	3	3
2	3	3	3	3
3	3	3	3	3

A\B	0	1	2	3
0	3	3	3	3
1	3	1	3	3
2	3	3	3	3
3	3	3	3	3

A\B	0	1	2	3
0	3	3	3	3
1	3	3	3	3
2	3	3	2	3
3	3	3	3	3

Existem também os postulados, por meio dos quais é possível desenvolver métodos de minimização de circuitos.

Postulado 1 *Identidade*
$a_1 + 0 = a_1$

Postulado 2 *Elemento Nulo*
$a_1 + L = L$
$a_1 *^i 0 = 0$

Postulado 3 *Idempotência*
$a_1 + a_1 = a_1$

Postulado 4 *Comutatividade*
$a_1 + a_2 = a_2 + a_1$
$a_1 *^i a_2 = a_2 *^i a_1$

Postulado 5 *Associatividade*
$a_1 + (a_2 + a_3) = (a_1 + a_2) + a_3$
$a_1 *^i (a_2 *^i a_3) = (a_1 *^i a_2) *^i a_3$

Postulado 6 *Complemento*
$a_1^0 + a_1^1 + ... + a_1^L = L$
$a_1 *^i a_1 *^i ... *^i a_1^L = 0$

Postulado 7 *Redução*
$(a_1^p *^i a_2^0) + (a_1^p *^i a_2^1) + ... + (a_1^p *^i a_2^L) = a_1^p *^i i$

Postulado 8 *Unicidade*
$i *^i i = i$

Postulado 9 *Involução*
$a_i^L = a_i$

Além disso, há o Teorema 1, que permite a subdivisão de um mapa em diferentes tabelas (F_1, F_2, F_3), de modo a extrair implicantes de cada submapa, gerando expressões para cada função, que serão por fim ligadas através do operador máximo.

Teorema 1 *Para a menor função F_i da função a ser sintetizada $G(a_1, ..., a_N)$, todos os mintermos que pertençam a uma função de ordem superior (a ordem é dada pelo valor do índice $_i$) são mintermos don't care para a função F_i, e todos os mintermos que pertençam a uma função de ordem inferior possuem valor 0.*

Um implicante pode ser definido como um agrupamento de mintermos, que são as células do mapa. É uma expressão que é verdadeira para o menor número de combinações de entradas. De modo a minimizar essa cobertura, utiliza-se a operação mais restritiva ($eAND_i$). Dessa forma, pode-se construir uma expressão que é verdadeira para apenas uma combinação dos valores de entrada. [19]

Assim, um implicante é um produto entre variáveis, ou seja, uma combinação de mintermos adjacentes. A simplificação de um circuito lógico é

a soma entre esses produtos, de modo que não se possa mais reduzir a expressão. Existe também a função maxtermo, a qual utiliza 0s para identificar implicantes na forma canônica POES [20], mas este trabalho irá utilizar apenas a forma canônica SOEP.

O mapa de Karnaugh estendido para o domínio quaternário assume o formato da Figura 1 para três variáveis.

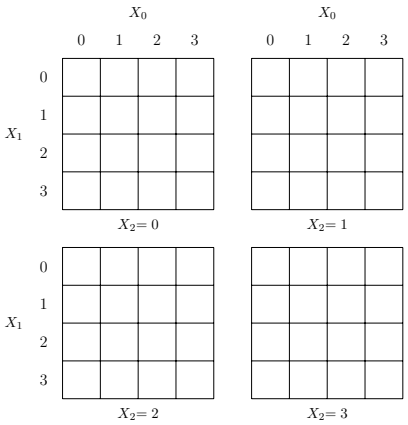

Figura 1: Mapa de Karnaugh quaternário com três variáveis.

3 Metodologia

Faz-se necessário, de modo a realizar o mapeamento bijetivo nas portas lógicas reversíveis, adicionar operandos de entrada e saída, bem como realizar a conexão adequada entre estes. Já com relação ao mapa de Karnaugh, é necessário encontrar uma nova estrutura básica de modo a permitir a aplicabilidade do método gráfico, bem como facilitar a concepção de um algoritmo.

3.1 Portas Lógicas Reversíveis

O mapeamento bijetivo para permitir computação direta e reversa depende da quantidade de operandos de cada porta lógica. Assim, para o operador SUC_r, que possui uma entrada (Vin_A) e uma saída (SUC_r), adiciona-se um operando auxiliar, chamado ancillary (Vin_C), e uma saída extra, nomeada garbage (g_{Ar}). Já aos operadores $eAND_{ir}$ e MAX_r, que possuem duas entradas (Vin_A e Vin_B) e uma saída ($eAND_{ir}$ e MAX_r, respectivamente), adiciona-se uma entrada ancillary (Vin_C) e duas saídas garbage (g_{Ar} e g_{Br}).

Para controlar a computação direta, é feito (Vin_C) igual ao nível zero, enquanto a computação reversa é feita com (Vin_C) igual a um nível não zero. Para reverter a operação, deve-se conectar a saída do operador (SUC_r, $eAND_{ir}$ ou MAX_r) à Vin_C, Vin_A à g_{Ar} e Vin_B à g_{Br} (este último não existe no caso do operador unário SUC_r), tal como na Figura 2, que exemplifica esse cenário para o operador $eAND_{ir}$.

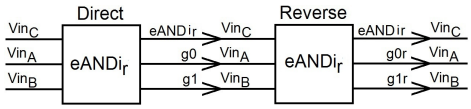

Figura 2: Exemplo de conexão para as operações direta e reversa.

3.2 Expansão do Mapa de Karnaugh

De modo a resolver o problema de se utilizar os mapas de Karnaugh para um número elevado de variáveis, propõe-se empilhar o mapa original, de modo a formar um cubo, que se tornará a nova estrutura básica, tal qual na Figura 3.

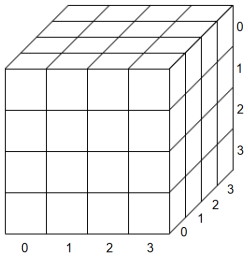

Figura 3: Estrutura com três variáveis no domínio quaternário.

Assim, a cada variável adicionada, o número de cubos é duplicado (para o domínio binário) ou quadruplicado (para o domínio quaternário). Implicantes então são extraídos separadamente em cada cubo, de modo a obter a melhor solução local. Posteriormente, esses implicantes são comparados com outros cubos que contenham mintermos logicamente adjacentes ao cubo em questão. Se possuírem rigorosamente a mesma cobertura de mintermos, ou seja, forem implicantes em iguais posições nos cubos verificados, então é possível juntar os implicantes encontrados, de forma a realizar uma melhor minimização da estrutura.

A verificação de adjacência é feita a partir do mapeamento de cubos de ordem superior aos de ordem inferior. Assim, dado um mintermo, é possível saber qual sua posição equivalente no cubo base através de (1), (2) e (3).

$$PositionX : Gray\ (Minterm \bmod 4) \tag{1}$$

$$PositionY : Gray\ (\left\lfloor \frac{Minterm}{4} \right\rfloor \bmod 4) \tag{2}$$

$$PositionZ : Gray\ (\left\lfloor \frac{Minterm}{16} \right\rfloor \bmod 4) \tag{3}$$

Considerando a identificação correta desses possíveis implicantes intercubos, é preciso verificar se os mintermos são de fato adjacentes, o que permitiria extrair um implicante maior. Isso é feito por meio da operação lógica disjunção exclusiva (XOR). Se identificado que apenas 1 bit varia entre o mintermo verificado e seu mapeamento no cubo base, então conclui-se que são adjacentes, e portanto os cubos de que fazem parte, com seus respectivos implicantes, também são.

O cubo base possui 6 variáveis, que são os bits menos significativos de cada mintermo, quando escritos em forma binária ($z_2z_1y_2y_1x_2x_1$). Caso na comparação entre os mintermos ocorra a variação de mais de 1 bit, então deve-se procurar qual é o cubo adjacente ao que está sendo avaliado realizando uma nova operação XOR que garanta essa adjacência, que é representada pela diferença de apenas 1 bit dentre os mais significativos (ou seja, quaisquer que não sejam os 6 primeiros).

4 Resultados

Após adaptação das portas lógicas e dos mapas de Karnaugh para adequarem-se, respectivamente, à reversibilidade e ao método de minimização proposto, surgem os seguintes resultados.

4.1 Operadores Lógicos Reversíveis

Para criar o primeiro subsistema, os diagramas de bloco são projetados utilizando as seguintes portas lógicas: inversor (INV), $NAND$ e NOR, e estas são representadas com um número dentro que denota a tensão de threshold (Vth), utilizada para descriminar os níveis lógicos quaternários, conforme a Figura 4. Se alguma porta não possuir número dentro, então o Vth é 1,5V. Tomando como exemplo a porta $INV_{0,7V}$ discrimina o nível lógico x = 0 dos níveis y = 1, 2, 3. $INV_{1,4V}$ discrimina os níveis lógicos x = 0, 1 dos níveis y = 2, 3, enquanto $INV_{2,2V}$ discrimina os níveis lógicos x = 0,

1, 2 do nível y = 3, de acordo com o Algoritmo 1. Aplica-se lógica análoga para as demais portas.

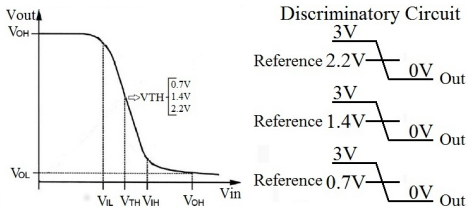

Figura 4: Tensão de threshold (Vth) discrimando os níveis lógicos quaternários.

Algoritmo 1 INV_{Vth}

if $VinA \leq Vth$ **then**
    x
else
    y
end if

O segundo e o terceiro subsistemas podem ser observados nos circuitos, onde a Figura 5 refere-se ao operador SUC_r, com suas respectivas funções lógicas verificadas na Equação (4). A Figura 6 refere-se ao operador $eAND_{1r}$, cuja função lógica é determinada pela Equação (5). A Figura 7 refere-se ao operador $eAND_{2r}$, tendo como funções lógicas as determinadas pela Equação (6). A Figura 8 refere-se ao operador $eAND_{3r}$, com a função lógica vista na Equação (7), e a Figura 9 refere-se ao operador MAX_r, com as funções lógicas da Equação (8).

Em particular para o operador SUC_r, devido ao já elevado número de transistores no domínio quaternário, quando se tem uma aplicação na qual é necessário implementar vários sucessores, é melhor fazer modificações na hora de setar o valor correto de saída na porta original (fazendo assim SUC_{2r}, SUC_{3r}) do que colocar vários SUC_{1r} em cascata.

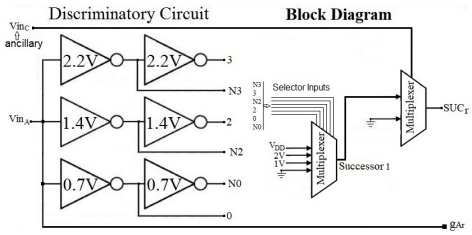

Figura 5: Circuito para o operador SUC_r.

$$\begin{aligned} 0 &= INV_{0.7V}(V_{in}A) \\ N0 &= INV_{0.7V}[INV_{0.7V}(V_{in}A)] \\ 2 &= INV_{1.4V}[INV_{1.4V}(V_{in}A)] \\ N2 &= INV_{1.4V}(V_{in}A) \\ 3 &= INV_{2.2V}[INV_{2.2V}(V_{in}A)] \\ N3 &= INV_{2.2V}(V_{in}A) \end{aligned} \tag{4}$$

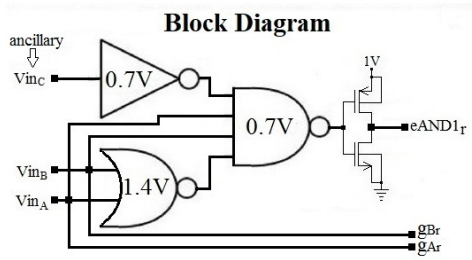

Figura 6: Circuito para o operador $eAND_{1r}$.

$$eAND1r = INV\{NAND_{0.7V}[INV_{0.7V}(V_{in}C), V_{in}A, V_{in}B, NOR_{1.4V}(V_{in}A, V_{in}B)]\} \tag{5}$$

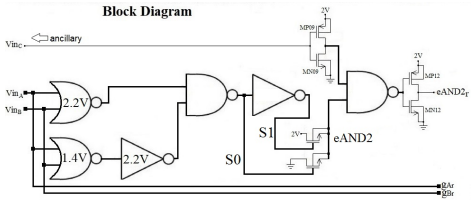

Figura 7: Circuito para o operador $eAND_{2r}$.

$$S0 = NAND\{NOR_{2.2V}(V_{in}A, V_{in}B), INV_{2.2V}[NOR_{1.4V}(V_{in}A, V_{in}B)]\}$$
$$S1 = INV(S0) \quad (6)$$

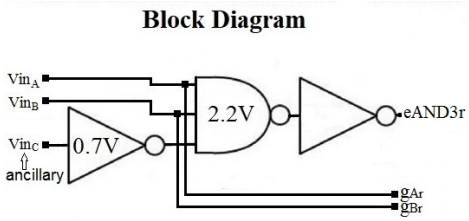

Figura 8: Circuito para o operador $eAND_{3r}$.

$$eAND_{3r} = INV\{NAND_{2.2V}[V_{in}A, V_{in}B, INV_{0.7V}(V_{in}C)]\} \quad (7)$$

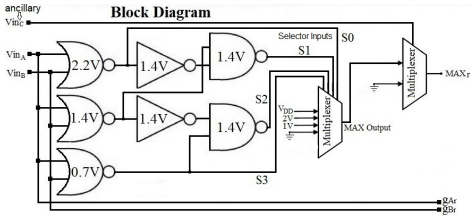

Figura 9: Circuito para o operador MAX_r.

$$S0 = NOR_{2.2V}(V_{in}B, V_{in}A)$$
$$S1 = NAND_{1.4V}\{NOR_{1.4V}(V_{in}B, V_{in}A), INV_{1.4V}[NOR_{2.2V}(V_{in}B, V_{in}A)]\}$$
$$S2 = NAND_{1.4V}\{NOR_{0.7V}(V_{in}B, V_{in}A), INV_{1.4V}[NOR_{1.4V}(V_{in}B, V_{in}A)]\}$$
$$S3 = NOR_{0.7V}(V_{in}B, V_{in}A) \quad (8)$$

4.2 Método de Minimização para o Domínio Binário

Considerando inicialmente a caminhada para a direita (eixo X): se a posição atual [x][y][z] possuir valor 1 e a próxima posição também, então a variável valueX, que representa a numeração em linha para os mintermos, é incrementada em uma unidade. Caso contrário, valueX é resetado, iniciando uma nova contagem a partir do próximo mintermo. O mesmo processo é repetido para as demais direções (eixo Y e eixo Z, respectivamente), até

que todos os mintermos estejam devidamente numerados. A partir desse processo, também são atribuídos valores às variáveis maxX, maxY e maxZ, que representam o maior número de 1's consecutivos em cada direção. O Algoritmo 2 demonstra como é feita essa numeração no eixo X, enquanto na Figura 10 verifica-se um exemplo de numeração em um mapa com quatro variáveis, onde os números nas extremidades de cada célula do mapa (X no canto inferior direito e Y no canto superior esquerdo) devem ser lidos de acordo com o formato *value (max)*.

Algoritmo 2 Numeração dos mintermos no eixo X

```
1:  valueX ← 1
2:  for k = 0 to axisZ − 1 do
3:      for j = 0 to axisY − 1 do
4:          for i = 0 to axisX − 1 do
5:              if minterm[k][j][i].binValue = 1 then
6:                  minterm[k][j][i].valueX ← valueX
7:                  valueX + +
8:              else
9:                  minterm[k][j][i].valueX ← 0
10:                 valueX ← 1
11:             end if
12:         end for
13:     end for
14: end for
```

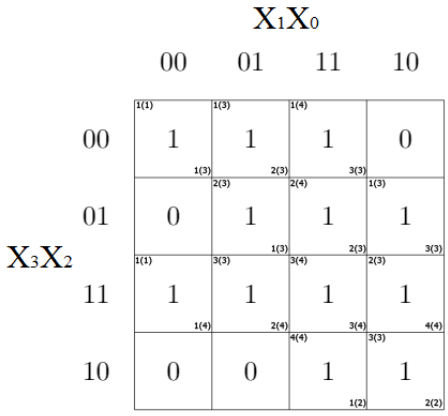

Figura 10: Mapa de Karnaugh no domínio binário após etapa de numeração.

Para a etapa de extração de implicantes, o ponto de partida deve ser o

último mintermo do mapa, seguindo em ordem decrescente, uma vez que mintermos de maior índice tendem a possuir valores mais altos para as variáveis *value* e *max*, o que permite uma simplificação na execução do algoritmo, evitando que muitas posições sejam analisadas. Definem-se assim as ranges para cada direção (rangeX, rangeY e rangeZ), de acordo com o Algoritmo 3, as quais serão comparadas com outros cubos. As ranges representam possíveis implicantes máximos locais, verificados por meio das numerações. Essa definição é feita a partir do eixo X e também a partir do eixo Y.

Algoritmo 3 Definição de ranges a partir do eixo X

```
1: for k = 0 to maxZ do
2:     for j = 0 to maxY do
3:         if maxX[z][y − j][x]! = 0 then
4:             if maxX[z][y − j][x] ≤ rangeX then
5:                 rangeX ← maxX[z][y − j][x]
6:             end if
7:             rangeY ← j + 1
8:         end if
9:     end for
10:    rangeX ← (2^floor(log2(rangeX)))
11:    rangeY ← (2^floor(log2(rangeY)))
12:    if maxZ[z − k][y][x] ≤ rangeZ then
13:        rangeZ ← maxX[z − k][y][x]
14:    end if
15:    rangeZ ← (2^floor(log2(rangeZ)))
16: end for
```

Isso porque no processo de identificação de implicantes primos, pode haver redundância quando se considera apenas uma direção de execução do algoritmo. Tomando uma outra direção e posteriormente comparando ambas, encontra-se a melhor solução local, contendo apenas implicantes primos essenciais, evitando assim a redundância. Tal possibilidade é abordada nas Figuras 11 e 12.

No processo de verificação de cubos adjacentes, deve-se procurar por ranges homólogas (ou seja, que cobrem mintermos nas mesmas posições dos outros cubos), de modo a obter um implicante de ordem maior, minimizando o circuito de modo mais eficiente.

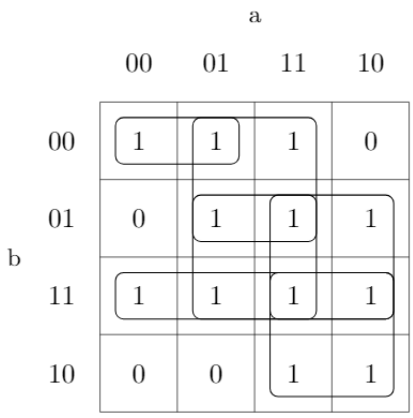

Figura 11: Implicantes via eixo X.

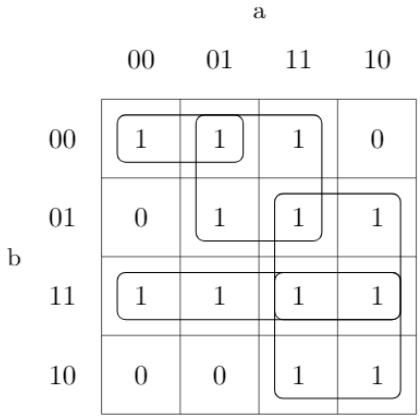

Figura 12: Implicantes via eixo Y.

Nota-se que no primeiro caso (via eixo X) há uma redundância, o implicante representado pelos mintermos (5,7,13,15). Isso demonstra a importância de tomar dois rumos de avaliação do mapa, de modo a obter uma minimização mais eficiente.

5 Discussão

Uma grande vantagem do método é que, a partir de modificações pontuais na estrutura, é possível alternar entre os domínios binário e quaternário. A cada variável adicionada no domínio binário, a quantidade de cubos dobra, enquanto no domínio quaternário essa replicação ocorre numa taxa 4^n. Além disso, o código Gray é necessário ao domínio binário, de modo a deixar os mintermos física e logicamente adjacentes, adaptação que não é necessária no domínio quaternário, visto que implicantes de 2 mintermos não existem nessa base.

5.1 Minimização de Circuitos Lógicos Quaternários

Uma etapa intermediária da minimização no domínio quaternário é a subdivisão do mapa para cada uma das subfunções ($F_1(a,b,c)$, $F_2(a,b,c)$ e $F_3(a,b,c)$), onde todos os mintermos de ordem inferior da função são substituidos por 0 e os de ordem superior são substituídos por *don't care*, de acordo com o Teorema 1.

A Figura 13 traz um exemplo de mapa de Karnaugh quaternário com 3 variáveis. Aplicando a ideia de subdivisão, tem-se o mapa da Figura 14 representando $F_1(a,b,c)$, o mapa da Figura 15 representando $F_2(a,b,c)$ e o

mapa da Figura 16 representando $F_3(a, b, c)$.

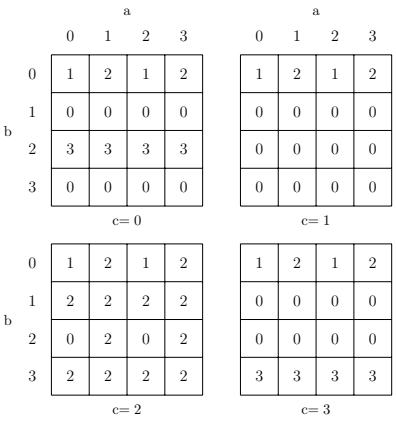

Figura 13: Exemplo de mapa de Karnaugh quaternário com três variáveis.

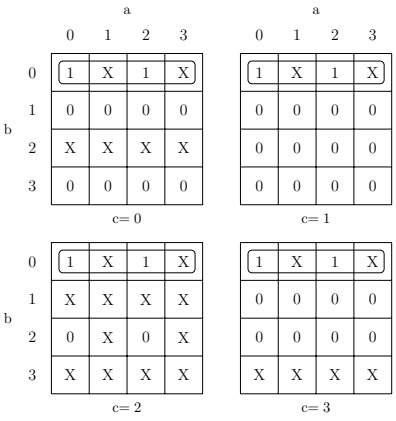

Figura 14: Mapa de $F_1(X_0, X_1, X_2)$ após a extração de implicantes.

Após análise dos mapas é possível realizar a extração dos implicantes:

$$F_1(a, b, c) = b^1 \tag{9}$$

$$F_2(a, b, c) = a^1 *^2 c + a^3 *^2 c + b^1 *^2 c + b^3 *^2 c + a^2 *^2 b^1 + a^2 *^2 b^3 \tag{10}$$

$$F_3(a, b, c) = b^1 *^3 c^3 + b *^3 c \tag{11}$$

A expressão final que representa a minimização do problema em questão é a união de F_1, F_2 e F_3 por meio do operador máximo.

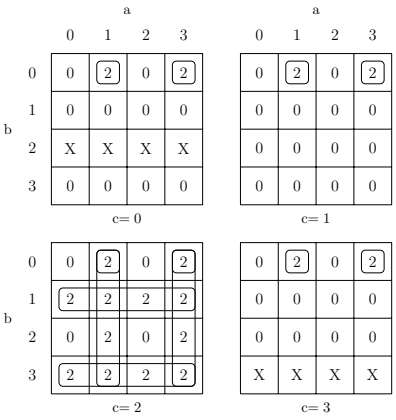

Figura 15: Mapa de $F_2(X_0, X_1, X_2)$ após a extração de implicantes.

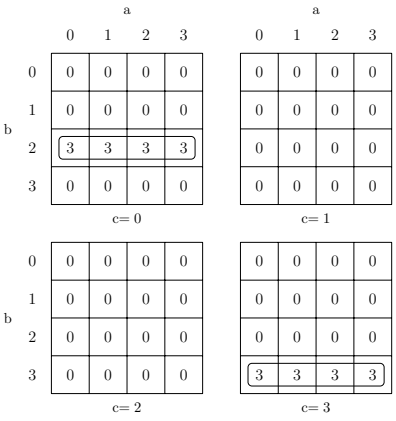

Figura 16: Mapa de $F_3(X_0, X_1, X_2)$ após a extração de implicantes.

6 Considerações Finais

O resultado do trabalho foi, baseando-se na álgebra quaternária proposta, a sugestão de uma metodologia de minimização para o domínio binário e extensível ao quaternário, bem como o projeto de portas lógicas reversíveis. Existem diferentes definições para estas portas. A escolhida é a mais simples, que permite implementar qualquer circuito na lógica quaternária com uma quantidade baixa de transistores Também é possível usar as saídas dos operadores, bem como adotar também os valores 1, 2 e 3 para a entrada V_{inC} para definir algum outro operador na saída, mas isso aumentaria bastante a complexidade do circuito sem grandes vantagens (por exemplo, em muitos momentos, determinada porta lógica seria utilizada apenas por um de seus operadores, e não pelo conjunto). Ainda assim é possível otimizar

as portas lógicas reversíveis atuais, de modo a reduzir a quantidade de gates necessárias, o que será feito em trabalhos futuros.

Quanto ao método, este é de fácil replicação ao longo dos eixos em ambos os domínios (binário e quaternário), mesmo que no primeiro demande a utilização do código Gray para manter os cubos física e logicamente adjacentes. Porém, mesmo a proposta de expansão ainda depende da habilidade humana de reconhecer padrões. Portanto, um código capaz de realizar todos os passos da minimização está em desenvolvimento, e sua finalização se dará em trabalhos futuros, para que haja tempo hábil para que seja testado e otimizado adequadamente. O passo de numeração do algoritmo não trata da tomada de implicantes de maneira cíclica, ou seja, entre as extremidades do cubo. Isso pode ser resolvido com o uso de templates que contemple esses casos. Além disso, a minimização deve ser aliada às portas lógicas reversíveis, substituindo os operadores SUC, $eAND_i$ e MAX por SUC_r, $eAND_{ir}$ e MAX_r, de modo a resolver ambos os problemas listados: reduzir o tamanho, os custos de implementação e a dissipação de energia dos chips. O resto do documento inclui o artigo publicado, e em anexo está o código em linguagem de programação C para a minimização do cubo base de 64 mintermos.

Referências

[1] S. L. Hurst, "Multiple-valued logic? its status and its future," *IEEE Transactions on Computers*, no. 12, pp. 1160–1179, 1984.

[2] C. Lazzari, P. Flores, and J. C. Monteiro, "Power and delay comparison of binary and quaternary arithmetic circuits," in *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on.* IEEE, 2009, pp. 1–6.

[3] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.

[4] V. Kabanets and J.-Y. Cai, "Circuit minimization problem," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 73–79.

[5] J. Łukasiewicz, "On three-valued logic," *The Polish Review*, pp. 43–44, 1968.

[6] E. L. Post, "Introduction to a general theory of elementary propositions," *American journal of mathematics*, vol. 43, no. 3, pp. 163–185, 1921.

[7] D. Kozen, "On kleene algebras and closed semirings," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1990, pp. 26–47.

[8] T. Toffoli, "Reversible computing," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1980, pp. 632–644.

[9] C. H. Bennett, "Logical reversibility of computation," *IBM journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973.

[10] M. E. Romero, E. M. Martins, R. R. dos Santos, and M. E. D. Gonzalez, "Universal set of cmos gates for the synthesis of multiple valued logic digital circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 3, pp. 736–749, 2013.

[11] K. C. Smith, "The prospects for multivalued logic: A technology and applications view," *IEEE Transactions on Computers*, no. 9, pp. 619–634, 1981.

[12] V. P. KS and K. Gurumurthy, "Quaternary cmos combinational logic circuits," in *2009 International Conference on Information and Multimedia Technology*. IEEE, 2009, pp. 538–542.

[13] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM journal of research and development*, vol. 5, no. 3, pp. 183–191, 1961.

[14] H. Thapliyal, M. Srinivas, and M. Zwolinski, "A beginning in the reversible logic synthesis of sequential circuits," 2005.

[15] A. S. M. Sayem and M. Ueda, "Optimization of reversible sequential circuits journal of computing," 2010.

[16] S. K. S. Hari, S. Shroff, S. N. Mahammad, and V. Kamakoti, "Efficient building blocks for reversible sequential circuit design," in *2006 49th IEEE International Midwest Symposium on Circuits and Systems*, vol. 1. IEEE, 2006, pp. 437–441.

[17] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers,*

Part I: Communication and Electronics, vol. 72, no. 5, pp. 593–599, 1953.

[18] P. K. Lala, *Principles of modern digital design*. John Wiley & Sons, 2007.

[19] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw Hill, 1994, no. BOOK.

[20] M. E. Romero, E. M. Martins, and R. R. Santos, "Multiple valued logic algebra for the synthesis of digital circuits," in *2009 39th International Symposium on Multiple-Valued Logic*. IEEE, 2009, pp. 262–267.

International Journal of Computer Applications (0975 - 8887)
Volume 174 - No.27, March 2021

Universal Set of Reversible Quaternary Logic Gates

Milton Ernesto Romero Romero
University of Mato Grosso do Sul
Campo Grande-Brazil CEP 7907-900

Diogo Anache de Souza
University of Mato Grosso do Sul
Campo Grande-Brazil CEP 79070-900

Evandro Mazina Martins
University of Mato Grosso do Sul
Campo Grande-Brazil CEP 79070-900

ABSTRACT

Reversible computing is of great interest due to the fact that the next generation of high performance computers must decrease heat dissipation in order to be practical, and irreversible gates dissipate energy into the environment because of the loss of information. This paper takes advantage of Multiple Valued Logic (MVL) quaternary universal set, that reduces integrated circuits (IC) interconnections, decreasing IC area, and with reversible gates that minimizes IC dissipation. The reversible computation permits both forward and backward computations, keeping the information entropy constant and decreasing heat dissipation, according to Landauer principle. The reversible gates are designed as an extension of the set of gates: Extended AND (eAND$_i$: eAND$_1$, eAND$_2$, eAND$_3$), Maximum (MAX) and Successor (SUC) already proposed in the literature. The voltage mode gates are implemented by means of three cascaded subsystems: the first subsystem discriminates 0,1,2,3 logical levels; the second subsystem performs the logic to implement each operator functionality; and the third subsystem set the right voltage output corresponding to 0,1,2,3 logical levels. Simulations with only 25, 18, 32, 10 and 32 CMOS transistors, respectively, utilizing Austriamicrosystems^{TM} technology with Cadence Virtuoso^{TM} tool demonstrate correct circuit behavior. These implementations present, for the irreversible circuits presented in the literature, fewer number of transistors.

Keywords

Reversible Computing, Universal Quaternary Set, Multiple Valued Logic

1. INTRODUCTION

Power consumption is a fundamental factor that must be addressed to build high performance systems. Among the possible paths to cope with this issue are: quantum (reversible) computing, classic reversible computing, nanotechnology, dark silicon concept, Multiple Valued Logic (MVL).

Quantum computing first discussed by Feynman [1] leads to the utilization of superposition and entanglement to perform reversible computation. In [2] the authors make a design of quantum Feynman and Toffoli gates with analysis of energy dissipation and in [3] an MVL circuit using Fredkin gates as a computational circuit suitable for reversible quantum computing is proposed. In [4] the synthesis of reversible gates in sequential circuits is proposed and nanotechnology is addressed in [5, 6] and the dark silicon concept is discussed in [7].

Quantum or classic reversible computing permits both forward and backward computations and aids to decrease energy dissipation by keeping constant entropy. In [8–11] the reversible computation is addressed. The optimization in reversible sequential circuits is proposed in [12, 13] and a methodology to the design of reversible circuits is shown in [14]. In [15] the reversible logic is demonstrated with adiabatic CMOS transistors and in [16] the reversible logic with the minimum of garbage signals is shown. In [17] the reversible logic is proposed using the adiabatic logic. Additional developments, addressed in [18], show the synthesis of reversible circuits based on Exclusive OR gate sum of products and [19] shows the implementation of reversible gate using the transistor with XOR Gate. An implementation at the transistor level for reversible digital circuit is found in [20] and [21] demonstrates a new reversible 2:4 decoder design. Design, synthesis, applications and state of the art in reversible gates are illustrated in [22, 23].

Multiple Valued Logic (MVL) allows the synthesis of digital circuits by increasing the domain to quaternary $D = \{0, 1, 2, 3\}$ digital representation. The MVL idea was introduced by Post [24] and Lukasiewicz [25] for the ternary algebra, with further developments in [26] that discuss the status of the MVL. Note that a quaternary digit corresponds to two binary bits. MVL decreases the number of interconnections, pads and power consumption, as well as the total area of the Integrated Circuit (IC), as the interconnections are about 70% of the total area [27, 28]. In [29] the MVL universal set of operators for any B base, that is, an algebra, minimization tools and synthesis methodology, is presented. In [30] a universal set of CMOS ports for the synthesis of digital logic circuits of multiple values is presented and a quaternary analog to digital converter application is addressed in [31]. Design based on ternary logic can be seen in [32].

All the above technologies help to cope with the energy issue and, in this environment, the purpose of this work is to demonstrate functional correctness and CMOS circuits feasibility through simulations of the combination of the MVL technique with classical reversible gates. Therefore, this work proposes the design and simulation of quaternary reversible gates that keeps the information entropy constant in the system, implying a bijective mapping between input and output, that decreases heat dissipation, according to Landauer principle [33]. This is due to the fact that, for one bit loss of information, that happens in each AND, OR gate of the combinational circuits, $KTln2$ Joules of energy are dissipated, where K stands for Boltzmann's constant (1.3807 $\times$ 10^{-23} Joules per Kelvin) and T is the absolute temperature, ln is the natural logarithm and the number 2 comes from the binary base. The non-reversible universal quaternary set of gates, already presented in the literature: Successor (SUC), Extended

AND ($eAND_1$, $eAND_2$, $eAND_3$), and Maximum (MAX) [30] is further extended to include the reversible characteristic by means of a bijective mapping that performs forward and backward computation with the same gate. Among the many ways to define the bijective mapping it is here presented one alternative. The actual implementation is based on three subsystems the first one discriminates each logical level, the second performs the logical steps to implement the gate under consideration and the third set the logical output level. Simulations with CMOS transistors utilizing Austriamicrosystems™ technology with Cadence Virtuoso™ tool demonstrate feasibility of the circuits and correct reversible quaternary computing behavior, additionally, the non-reversible gates have fewer number of transistors (only 25, 18, 32, 10 and 32 MOS transistors, for the $eAND_i$, MAX, SUC, respectively) that is better than [30] in terms of transistors counting.

The rest of this paper is organized as follows: Section 2 defines the MVL reversible primal algebra; Section 3 addresses the MVL operators CMOS implementation; Section 4 presents results and the discussion and finally; Section 5 summarizes the concluding remarks and future work.

2. MVL REVERSIBLE PRIMAL ALGEBRA

For notation purposes SUC, $eAND_i$, MAX [29] and SUC_r, $eAND_ir$, MAX_r denote non-reversible and reversible gates defined here, respectively. SUC is an unary (i.e. one operand) operator and $eAND_i$, MAX are binary operators (i.e. two operands).

The non-reversible universal quaternary set of gates is defined in the cyclic ordered domain D:{0,1,2,3} as follows:

Let i, inputs: VinA, VinB, and the outputs: SUC, $eAND_i$, MAX $\in$ D:{0,1,2, 3}.

$eAND_i$ definition: if VinA=VinB=i then $eAND_i$=i, otherwise $eAND_i$=0.

MAX definition: if VinA$\geq$VinB then MAX=VinA, otherwise MAX=VinB.

SUC definition: if VinA=i then SUC=(i+1) Mod 4. Where Mod stands for the Modulo operation.

There are many ways to define the bijective mapping that performs forward and backward computation and it is up to the designer to choose one.

For two inputs reversible operators implementations, as shown in Tables 1, 2, 3, 4 there exist three inputs: two operator inputs (VinA,VinB) and one ancillary input (VinC); and three outputs: the operator output ($eAND_{ir}$ or MAX_r) and two garbage outputs (g_{Br},g_{Ar}). For one input reversible operator implementation, as shown in Table 5 there exists two inputs: one operator input (VinA) and one ancillary input (VinC) and two outputs: one operator output (SUC_r) and one garbage output (g_{Ar}).

Table 1. eAND$_{3r}$ direct → inverse

Vin$_C$	Vin$_B$	Vin$_A$	eAND3$_r$	g$_{Br}$	g$_{Ar}$		Vin$_C$	Vin$_B$	Vin$_A$	eAND3$_r$	g$_{Br}$	g$_{Ar}$
ancillary							eAND3$_r$	g$_{Br}$	g$_{Ar}$	Vin$_C$	Vin$_B$	Vin$_A$
0	0	0	0	0	0	→	0	0	0	0	0	0
0	0	1	0	0	1	→	0	0	1	0	0	1
0	0	2	0	0	2	→	0	0	2	0	0	2
0	0	3	0	0	3	→	0	0	3	0	0	3
0	1	0	0	1	0	→	0	1	0	0	1	0
0	1	1	0	1	1	→	0	1	1	0	1	1
0	1	2	0	1	2	→	0	1	2	0	1	2
0	1	3	0	1	3	→	0	1	3	0	1	3
0	2	0	0	2	0	→	0	2	0	0	2	0
0	2	1	0	2	1	→	0	2	1	0	2	1
0	2	2	0	2	2	→	0	2	2	0	2	2
0	2	3	0	2	3	→	0	2	3	0	2	3
0	3	0	0	3	0	→	0	3	0	0	3	0
0	3	1	0	3	1	→	0	3	1	0	3	1
0	3	2	0	3	2	→	0	3	2	0	3	2
0	3	3	0	3	3	→	0	3	3	0	3	3

Table 2. eAND$_{2r}$ direct → inverse

Vin$_C$	Vin$_B$	Vin$_A$	eAND2$_r$	g$_{Br}$	g$_{Ar}$		Vin$_C$	Vin$_B$	Vin$_A$	eAND2$_r$	g$_{Br}$	g$_{Ar}$
ancillary							eAND2$_r$	g$_{Br}$	g$_{Ar}$	Vin$_C$	Vin$_B$	Vin$_A$
0	0	0	0	0	0	→	0	0	0	0	0	0
0	0	1	0	0	1	→	0	0	1	0	0	1
0	0	2	0	0	2	→	0	0	2	0	0	2
0	0	3	0	0	3	→	0	0	3	0	0	3
0	1	0	0	1	0	→	0	1	0	0	1	0
0	1	1	0	1	1	→	0	1	1	0	1	1
0	1	2	0	1	2	→	0	1	2	0	1	2
0	1	3	0	1	3	→	0	1	3	0	1	3
0	2	0	0	2	0	→	0	2	0	0	2	0
0	2	1	0	2	1	→	0	2	1	0	2	1
0	2	2	2	2	2	→	2	2	2	0	2	2
0	2	3	0	2	3	→	0	2	3	0	2	3
0	3	0	0	3	0	→	0	3	0	0	3	0
0	3	1	0	3	1	→	0	3	1	0	3	1
0	3	2	0	3	2	→	0	3	2	0	3	2
0	3	3	0	3	3	→	0	3	3	0	3	3

Table 3. eAND$_{1r}$ direct → inverse

Vin$_C$	Vin$_B$	Vin$_A$	eAND1$_r$	g$_{Br}$	g$_{Ar}$		Vin$_C$	Vin$_B$	Vin$_A$	eAND1$_r$	g$_{Br}$	g$_{Ar}$
ancillary							eAND1$_r$	g$_{Br}$	g$_{Ar}$	Vin$_C$	Vin$_B$	Vin$_A$
0	0	0	0	0	0	→	0	0	0	0	0	0
0	0	1	0	0	1	→	0	0	1	0	0	1
0	0	2	0	0	2	→	0	0	2	0	0	2
0	0	3	0	0	3	→	0	0	3	0	0	3
0	1	0	0	1	0	→	0	1	0	0	1	0
0	1	1	1	1	1	→	1	1	1	0	1	1
0	1	2	0	1	2	→	0	1	2	0	1	2
0	1	3	0	1	3	→	0	1	3	0	1	3
0	2	0	0	2	0	→	0	2	0	0	2	0
0	2	1	0	2	1	→	0	2	1	0	2	1
0	2	2	0	2	2	→	0	2	2	0	2	2
0	2	3	0	2	3	→	0	2	3	0	2	3
0	3	0	0	3	0	→	0	3	0	0	3	0
0	3	1	0	3	1	→	0	3	1	0	3	1
0	3	2	0	3	2	→	0	3	2	0	3	2
0	3	3	0	3	3	→	0	3	3	0	3	3

For all operators, VinC controls direct (VinC equals to 0 level) or reverse computation (VinC equals to non 0 level). The outputs g_{Br}, g_{Ar} are utilized in order to not repeat any possible code making the mapping invertible. Therefore, they are not useful for any other purpose except the invertibility. Of course, it is possible to utilize these output to define other operators, but for the purpose in this work, there is not any utility to implement that and if implemented it would increment further the complexity of the electronic circuits. To control forward and backward computation with the same gate, they are connected as shown in Figure 1 in which VinC equals 0 level performs the direct and VinC equals to non 0 level performs the inverse computation. VinB, VinA are the operators' inputs. To reverse the operator each output of the direct gate computation:

Table 4. MAX$_r$ direct → inverse

Vin$_C$	Vin$_B$	Vin$_A$	MAX$_r$	g$_{Br}$	g$_{Ar}$		Vin$_C$	Vin$_B$	Vin$_A$	MAX$_r$	g$_{Br}$	g$_{Ar}$
ancillary							MAX$_r$	g$_{Br}$	g$_{Ar}$	Vin$_C$	Vin$_B$	Vin$_A$
0	0	0	0	0	0	→	0	0	0	0	0	0
0	0	1	1	0	1	→	1	0	1	0	0	1
0	0	2	2	0	2	→	2	0	2	0	0	2
0	0	3	3	0	3	→	3	0	3	0	0	3
0	1	0	1	1	0	→	1	1	0	0	1	0
0	1	1	1	1	1	→	1	1	1	0	1	1
0	1	2	2	1	2	→	2	1	2	0	1	2
0	1	3	3	1	3	→	3	1	3	0	1	3
0	2	0	2	2	0	→	2	2	0	0	2	0
0	2	1	2	2	1	→	2	2	1	0	2	1
0	2	2	2	2	2	→	2	2	2	0	2	2
0	2	3	3	2	3	→	3	2	3	0	2	3
0	3	0	3	3	0	→	3	3	0	0	3	0
0	3	1	3	3	1	→	3	3	1	0	3	1
0	3	2	3	3	2	→	3	3	2	0	3	2
0	3	3	3	3	3	→	3	3	3	0	3	3

Table 5. SUC$_r$ direct → inverse

Vin$_C$	Vin$_A$	SUCr	g$_{Ar}$		Vin$_c$	Vin$_a$	SUCr	g$_{Ar}$
ancillary					SUC$_r$	g$_{Ar}$	Vin$_C$	Vin$_A$
0	0	1	0	→	1	0	0	0
0	1	2	1	→	2	1	0	1
0	2	3	2	→	3	2	0	2
0	3	0	3	→	0	3	0	3

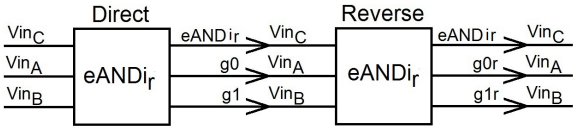

Fig. 1: eAND$_{ir}$ direct and eAND$_{ir}$ reverse block diagram connection

$eAND_{ir}$ or MAX_r or SUC_r is connected to the input VinC of the reverse gate computation, g_{Br} is connected to VinB and g_{Ar} is connected to VinA for the binary operators and only g_{Ar} is connected to VinA for the unary operator (SUC_r).

3. MVL OPERATORS CMOS IMPLEMENTATION

All CMOS operators' implementation description is based on three subsystems and is presented based on the Figures in block diagrams and CMOS gates circuits. In all Figures, at the top, the circuit that is focused on the gates' implementation at the input/output $voltage$ level along with the algorithms describing the gates' implementation, is shown. At the bottom, the block diagram that is focused on the input/output $logical$ level along with the logical equations describing them, is also shown. All gates utilize the discriminators (first subsystem) to verify the logical level input; the binary logic circuit (second subsystem) implements the logic of the particular level under consideration; and finally, the multiplexers or switches (third subsystem) set the output to the right voltage level. For example, the $eAND_3$ gate demands that both inputs are set to the 3 logical level. Then, the discriminators (first subsystem) verify these conditions, the binary logic circuit (second subsystem) verifies the intersection in the 3 level, that is, both inputs are set to the 3 logical level simultaneously; and finally, the multiplexers (third subsystem) set the output to the right voltage level (3V). All others gates follow the same design criteria, as presented next.

Note that the logical levels are defined as: (ground) 0V is the 0 logical level, 1V is the 1 logical level, 2V is the 2 logical level, (VDD) 3V is the 3 logical level. However, if in the middle of the circuit a particular gate output is set to 3V, it does not always mean the 3 logical level, it only means that the output is high voltage (because it is binary), by looking at the circuit gate and the block diagram the actual situation is clear by the context.

For the logic implementation the INV (inverter), NAND, NOR binary gates are utilized and they are represented as in the binary logic with a number inside that defines the threshold voltage (Vth) utilized to discriminate the quaternary logical levels. Whenever there is not any number inside, the Vth is set to 1.5V (middle of VDD=3V polarization voltage of the gate). In the following, two criteria are needed. First, for all gates, first subsystem discriminates which logical level (0,1,2,3) is at the input by comparing each input (VinC, VinB, VinA) with corresponding threshold Vth values (0.7V, 1.4V, 2.2V), as shown in Figure 2, defined by setting the CMOS width and length sizes.

The implementation utilizes binary levels, that is, all these gates always discriminate between two subsets, as for example, 0 level from 1,2,3 levels or 0,1 levels from 2,3 levels, etc. As shown in

Table 6. Algorithms table

Algorithm 1	Algorithm 2	Algorithm 3
INV$_{Vth}$	NAND$_{Vth}$	NOR$_{Vth}$
If VinA $\leqslant$ Vth	If (VinA AND VinB) $\leqslant$ Vth	If (VinA OR VinB) $\leqslant$ Vth
x	x	x
Else	Else	Else
y	y	y
EndIf	EndIf	EndIf

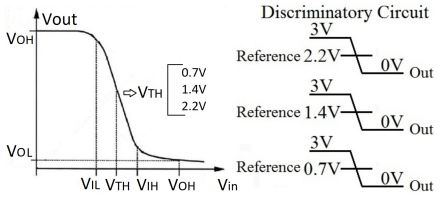

Fig. 2: MVL thresholds of the discriminators

the Algorithm 1 in Table 6, INV$_{0.7V}$(VinA) discriminates between subsets x=0 logical level from y=1,2,3 logical levels, when its input (VinA) is less than 0.7V (in the gate circuit, the output is 3V); INV$_{1.4V}$(VinA) discriminates between subsets x=0,1 logical levels from y=2,3 logical levels, when its input (VinA) is less than 1.4V (in the gate circuit, the output is 3V); INV$_{2.2V}$(VinA) discriminates between subsets x=0,1,2 logical levels from y=3 logical level, when its input (VinA) is less than 2.2V (in the gate circuit, the output is 3V); INV(VinA) discriminates to set 0V or VDD, when its input (VinA) is less than 1.5V (in the gate circuit, VDD is the polarization of the CMOS).

Second, the target of the discrimination for a given gate, i.e. what is the subset (x or y) of interest, as for example: x=0 or y=1,2,3; x=0,1 or y=2,3 and so on.

In the circuit description, in the block diagram, the logical function to implement the gate under consideration is presented, and after that, the CMOS circuit to show the actual voltages in the implementation.

For the presentation of the logical function, the name of the gate with its threshold (i.e INV$_{0.7V}$) is utilized as the name of the quaternary function corresponding to the NAND or NOR or INV in the block diagram and in the parenthesis its inputs, as shown latter.

Same criteria for the NAND and NOR binary gates in which the Vth helps to discriminate logical (x or y) levels subsets, detailed latter in the description of each operator implementation.

As shown in the Algorithm 2 in Table 6, NAND$_{Vth}$(VinB,VinA) discriminates between subsets x logical levels from y logical levels, when both of its inputs (VinB,VinA) are greater than Vth (in the gate circuit, the output is 0V, meaning that both inputs belongs to the y subset).

As shown in the Algorithm 3 in Table 6, NOR$_{Vth}$(VinB,VinA) discriminates between subsets x logical levels from y logical levels, when one of its inputs (VinB or VinA) are greater than Vth (in the gate circuit, the output is 0V, meaning that both inputs belongs to the x subset). Follows the actual gates implementation description.

3.1 $eAND_{ir}$ Implementations

3.1.1 $eAND_{3r}$ Implementation. The binary operator $eAND_{3r}$ implementation, as defined in Table 1 is shown in Figure 3. Note that columns VinB, VinA and $eAND_{3r}$ implement exactly the non-reversible $eAND_3$ operator definition. For the two operands $eAND_{3r}$ implementation, there exists three inputs (VinC, VinB, VinA) and three outputs ($eAND_{3r}$, g_{Br}, g_{Ar}). VinC (ancillary

Table 7. eAND$_{3r}$ transistors size

Transistor	W	L	Transistor	W	L
PMOS	(μm)	(μm)	NMOS	(μm)	(μm)
MP01	20	0.35	MN01	10	0.35
MP02/03	20	0.35	MN02/03	0.4	0.35
MP04	0.4	0.35	MN04	0.4	0.35
MP05	0.8	0.35	MN05	0.4	0.35

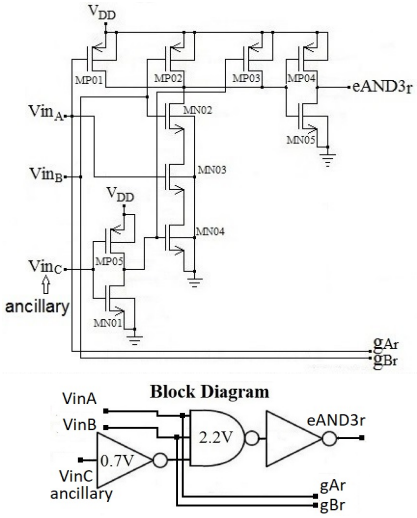

Fig. 3: eAND$_{3r}$ circuit

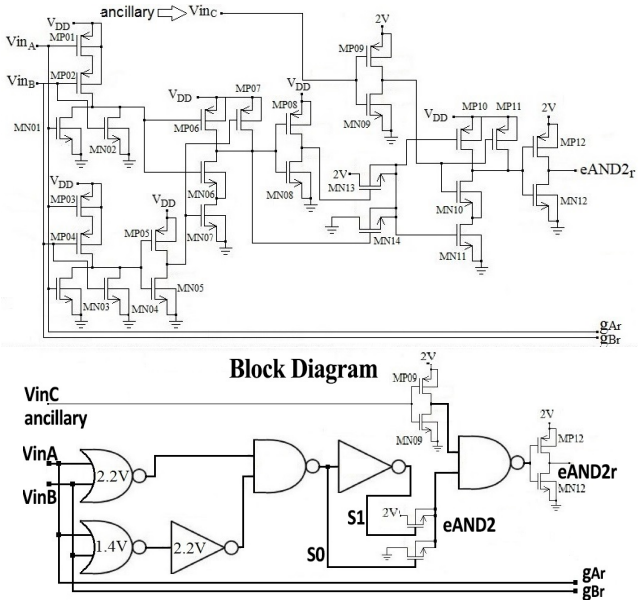

Fig. 4: eAND$_{2r}$ circuit

input) controls direct (VinC equals to 0 level) and to reverse the computation (VinC equals to non 0 level). Follows the logical function based on the operator circuit, in Equation 1.

$$eAND_{3r} = INV\{NAND_{2.2V}[VinA, VinB, INV_{0.7V}(VinC)]\} \quad (1)$$

As it can be seen in Figure 3, in this function the target of the INV with CMOS transistors (MN05,MP04) and threshold 1.5V with output $eAND_{3r}$ is to set the 3 level, when all three inputs of the NOR$_{2.2V}$ (VinA,VinB,INV$_{0,7V}$(VinC)) belong to the 3 level.
In the arguments of the function, the target of the INV$_{0.7V}$ with input VinC is to identify level 0 to control forward and backward computation with CMOS transistors (MN01,MP05). The target for the NAND$_{2.2V}$ is to identify that all inputs belong to the 3V with CMOS transistors (MN02,MP03,MN04,MP01,MP02, MN03), and therefore, this intersection defines that both VinA, VinB must have 3 logical levels. Table 7 shows the size of the CMOS transistors.

3.1.2 $eAND_{2r}$ Implementation. The binary operator $eAND_{2r}$ implementation, as defined in Table 2, is shown in Figure 4. Note that columns VinB, VinA and $eAND_{2r}$ implement exactly the non-reversible $eAND_2$ operator definition. For the two operands $eAND_{2r}$ implementation, there exists three inputs (VinC, VinB, VinA) and three outputs ($eAND_{2r}$, g$_{Br}$, g$_{Ar}$). VinC (ancillary input) controls direct (VinC equals to 0 level) and to reverse the computation (VinC equals to non 0 level). When the selector S1 equals to 2 level then $eAND2$=2V; otherwise $eAND2$=0V. Follows the logical functions based on the operator circuit, in Equation 2.

$$S1 = INV(S0) \qquad S0 = NAND\{NOR_{2.2V}(VinA, VinB), INV_{2.2V}[NOR_{1.4V}(VinA, VinB)]\} \quad (2)$$

As shown in the Figure 4 the INV with input VinC with polarization voltage of 2V controls forward (VinC=0V) and backward (VinC different from 0 level) computation with CMOS transistors (MN09,MP09).

Output $eAND_{2r}$ is set to 2 level by inverting the NAND gate if both inputs (direct computation controlled by VinC and $eAND_2$) are in the high level simultaneously, that is, $eAND_2$ output is set to 2V by activating the transistor (MN13); otherwise $eAND_{2r}$ is set to 0V by activating the transistor (MN14).
In the argument of the S0 function, NAND has two inputs: first the output of the NOR with threshold 2.2V (NOR$_{2,2V}$); second the output of the INV$_{2.2V}$ with input NOR$_{1.4V}$(VinB,VinA).
The target of the NAND is that both of its inputs are set in the 2 level by the intersection between the subsets 0,1,2 and 2,3 with CMOS transistors (MN06,MP06,MN07,MP07). The target of the NOR$_{2.2V}$ is to identify the subset 0,1,2 (setting its output to 3V) with CMOS transistors (MN01,MP01,MN02, MP02). The target for the NOR$_{1.4V}$ is to identify the subset 0,1 level (setting its output to 0V) with CMOS transistors (MN03,MP03,MN04,MP04). The next inverter INV$_{2.2V}$(NOR$_{1,4V}$(VinB,VinA)) inverts the signal to identify the subset 2,3 level (setting its output to 3V) with CMOS transistors (MN05, MP05). Table 8 shows the size of the CMOS transistors.

3.1.3 $eAND_{1r}$ Implementation. The binary operator $eAND_{1r}$ implementation, as defined in Table 3 is shown in Figure 5. Note that columns VinB, VinA and $eAND_{1r}$ implement exactly the non-reversible $eAND1$ operator definition. For the two operands $eAND_{1r}$ implementation, there exists three inputs (VinC, VinB, VinA) and three outputs ($eAND_{ir}$, g$_{Br}$, g$_{Ar}$). VinC (ancillary input) controls direct (VinC equals to 0 level) and to reverse the

Table 8. eAND$_{2r}$ transistors size

Transistor PMOS	W (μm)	L (μm)	Transistor NMOS	W (μm)	L (μm)
MP01/02	25	1	MN01/02	1	15
MP03/04	24	0.35	MN03/04	0.4	0.35
MP05/06/07/08	10	0.35	MN05/06/07/08	10	0.35
MP09	0.8	0.35	MN09	10	0.35
MP10/11/12	10	0.35	MN10/11/12	10	0.35
			MN13/14	2	2

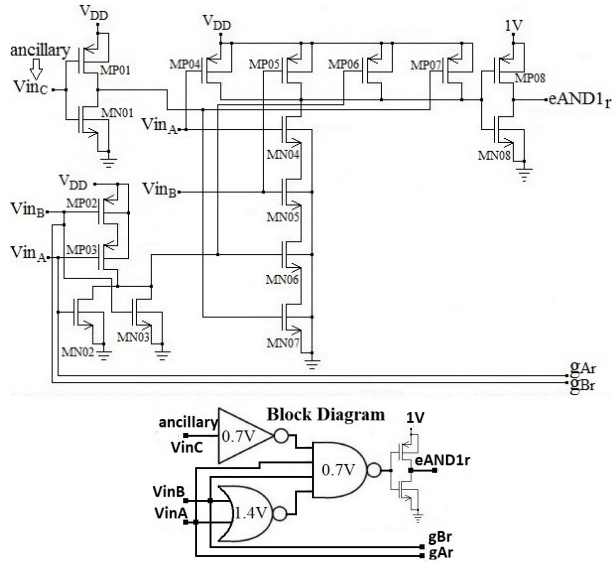

Fig. 5: eAND$_{1r}$ circuit

computation (VinC equals to non 0 level). Follows the logical functions based on the operator circuit, in Equation 3.

$$eAND_{1r} = INV\{NAND_{0.7V}[INV_{0.7V}(VinC), VinA, VinB, NOR_{1.4V}(VinA, VinB)]\} \quad (3)$$

In the argument of the $eAND_{1r}$ function, the INV with threshold 1.5V with CMOS transistors (MN08,MP08) with polarization voltage of 1V set the output to 1 level. the argument to the INV function is the (NAND$_{0.7V}$) with threshold 0.7V that has four inputs: first the output of the inverter with threshold 0.7V (INV$_{0.7V}$) that has as its input the VinC input; second the input VinA; third the VinB input; and fourth the output of the NOR with threshold 1.4V (NOR$_{1.4V}$) with inputs (VinB,VinA). The target of the INV$_{0.7V}$ it to identify level 0 to control forward and backward computation with CMOS transistors (MN01,MP01). The target of the NOR$_{1.4V}$ is to identify if both inputs belong to the 0,1 levels (NOR$_{1.4V}$ output in 3V) with CMOS transistors (MN02,MP02,MN03,MP03). The target for the NAND$_{0.7V}$ is to identify that all inputs belong to the 1,2,3 logical levels (NAND$_{0.7V}$ output in 0V) with CMOS transistors (MN04,MP04,MN05,MP05, MN06,MP06,MN07,MP07), and therefore, this intersection defines that both VinA, VinB must have 1 logical levels. The last inverter (MN08, MP08) set the $eAND_{1r}$ output to 1V, that is the $eAND_{1r}$ voltage level output. g$_{Ar}$, g$_{Br}$ are exactly the same as VinB and VinA, respectively. Table 9 shows the size of the CMOS transistors.

3.2 MAX_r Implementation

The binary operator MAX_r implementation, as defined in Table 4, is shown in Figure 6. Note that columns VinB, VinA and MAX_r implement exactly the non-reversible MAX operator definition.

Table 9. eAND$_{1r}$ transistors size

Transistor PMOS	W (μm)	L (μm)	Transistor NMOS	W (μm)	L (μm)
MP01	0.8	0.35	MN01	10	0.35
MP02/03	5	0.35	MN02/03	0.4	0.35
MP04/05/06/07	0.4	0.35	MN04/05/06/07	18	0.35
MP08	10	0.35	MN08	10	0.35

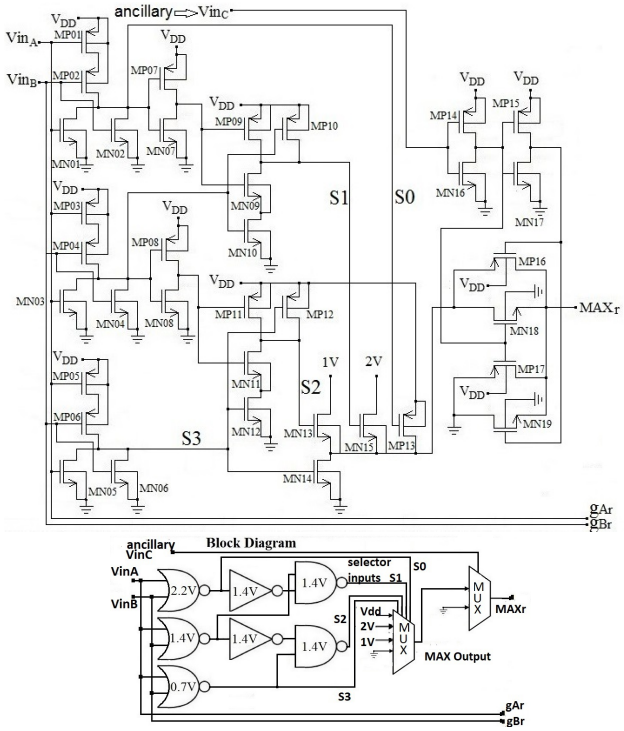

Fig. 6: MAX$_r$ circuit

For the two operands MAX_r implementation there exists three inputs, (VinC, VinB, VinA) and three outputs (MAX_r, g$_{Br}$, g$_{Ar}$). VinC (Ancillary) controls direct (VinC equals to 0 level) or reverse computation (VinC equals to non 0 level). VinB, VinA are the operators' inputs. g$_{Br}$, g$_{Ar}$ (garbage) are utilized in order to making the mapping invertible.
When VinC=0 level, the selector input of the multiplexer with output MAX_r and inputs (0V=ground and Max output) computes the forward computation; otherwise it computes the backward computation. Follows the logical functions based on the operator circuit, in Equation 4.

$$\begin{aligned} S0 &= NOR_{2.2V}(VinB, VinA) \\ S1 &= NAND_{1.4V}\{NOR_{1.4V}(VinB, VinA), \\ &\quad INV_{1.4V}[NOR_{2.2V}(VinB, VinA)]\} \\ S2 &= NAND_{1.4V}\{NOR_{0.7V}(VinB, VinA), \\ &\quad INV_{1.4V}[NOR_{1.4V}(VinB, VinA)]\} \\ S3 &= NOR_{0.7V}(VinB, VinA) \end{aligned} \quad (4)$$

The implementation criteria for the selector inputs to the quaternary multiplexer is that when one and only one of the selector inputs is different from 0 level the MAX output signal must be set to that level, otherwise it is set to the 0 level.
Therefore, if the selector input (S0=0V) generated by the NOR$_{2.2V}$ identifies the 3 level then, MAX output equals to 3 level, and all the other inputs in the selector inputs are 0 level. The target of the NOR$_{2.2V}$ is to discriminate x=0,1,2 level from y=3 level with CMOS transistors (MN01, MP01, MN02, MP02).
If the selector input (S1=3V) generated by the NAND$_{1.4V}$ identifies the 2 level then MAX output equals to 2 level, and all the other inputs in the selector inputs are 0 level. The target of the

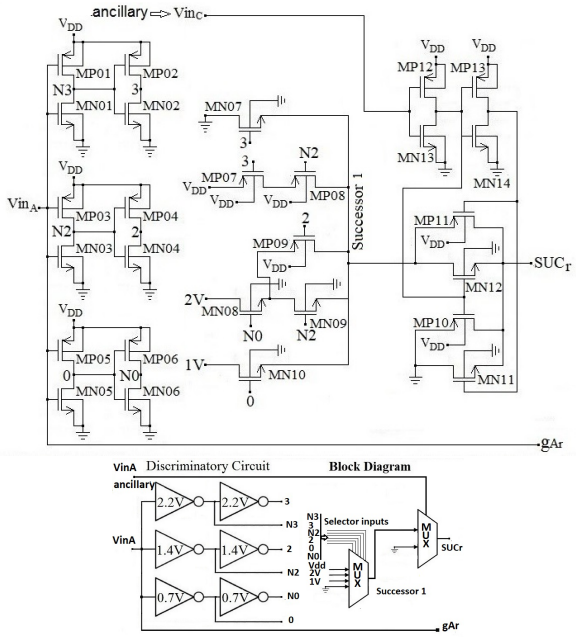

Fig. 7: SUC$_r$ circuit

NAND$_{1.4V}$ is to discriminate x=0,1 level from y=2,3 level with CMOS transistors (MN09, MP09, MN010, MP010). The target of the NOR$_{1.4V}$ is to discriminate x=0,1 level from y=2,3 level with CMOS transistors (MN03, MP03, MN04, MP04). INV$_{1.4V}$ inverts the signal with CMOS transistors (MN07, MP07).
If the selector input (S2=3V) generated by the NAND$_{1.4V}$ identifies the 1 level then MAX output is 1 level, and all the other inputs in the selector inputs are 0 level. The target of the NOR$_{1.4V}$ is to discriminate x=0,1 level from y=2,3 level with CMOS transistors (MN11, MP11, MN12, MP12). INV$_{1.4V}$ inverts the signal with CMOS transistors (MN08, MP08).
If the selector input (S3=3V) generated by the NOR$_{0.7V}$ identifies the 0 level then MAX output is 0 level, and all the other inputs in the selector inputs are 0 level. The target of the NOR$_{0.7V}$ is to discriminate x=0,1 level from y=2,3 level with CMOS transistors (MN05, MP05, MN06, MP06). Table 10 shows the size of the CMOS transistors.

3.3 SUC_r Implementation

The unary operator SUC_r implementation, as defined in Table 5 is shown in Figure 7. Note that columns VinA and SUC_r implement exactly the non-reversible SUC operator definition.
VinC inputs to the quaternary multiplexer selector with inputs Successor1 and ground (0 level) to control direct (VinC equals

Table 10. MAX$_r$ transistors size

Transistor PMOS	W (μm)	L (μm)	Transistor NMOS	W (μm)	L (μm)
MP01/02	10	0.35	MN01/02	0.4	10
MP03/04	4.4	0.35	MN03/04	0.5	0.35
MP05/06	0.4	0.35	MN05/06	10	0.35
MP07 up to 12	4.4	0.35	MN07 up to 12	2.8	0.35
MP13	5.9	2	MN13	1	1
MP14	0.8	0.35	MN14	1	1
MP15	10	0.35	MN15	1	1
MP16/17	5.9	2	MN16/17	10	0.35
			MN18/19	2	2

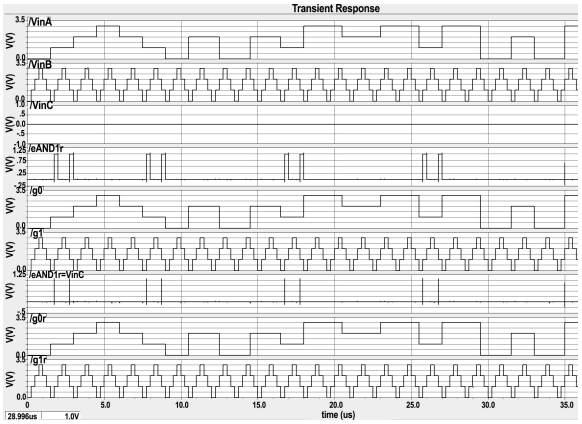

Fig. 8: eAND$_{1r}$ simulation

to 0 level, then SUC_r =Successor1) by activating the switch (MP11,MN12) to set SUC_r=Sucessor1 level through the transistors (MP12,MN13,MP13,MN14). To reverse the computation (VinC equals to non 0 level, then SUC_r=0 level) by activating the switch (MP10,MN11) to set SUC_r=0 level of the multiplexer.
The other quaternary multiplexer with signal inputs 3, N3, 2, N2, 0, N0 set Sucessor1=i as VinA=i level which level is at the input SUC_r, and then, the output SUC_r=i+1 level by means of choosing the right input from: ground (0 level), 1V (1 level), 2V (2 level), VDD (3 level). Follows the logical functions based on the operator circuit, in Equation 5.

$$\begin{aligned} 0 &= INV_{0.7V}(VinA) \quad & N0 &= INV_{0.7V}[INV_{0.7V}(VinA)] \\ 2 &= INV_{1.4V}[INV_{1.4V}(VinA)] \quad & N2 &= INV_{1.4V}(VinA) \\ 3 &= INV_{2.2V}[INV_{2.2V}(VinA)] \quad & N3 &= INV_{2.2V}(VinA) \end{aligned} \tag{5}$$

Signal 3: INV$_{2,2V}$(VinA) discriminates x=0,1,2 levels from y=3 level, with the CMOS transistors (MN01,MP01). The target is the y=3 level. Then, it identifies if VinA is in the level 3 and N3 identifies that is in the x=0,1,2 levels (N3=Not 3), with the CMOS transistors (MN02,MP02). Signal 2: INV$_{1,4V}$(VinA) discriminates x=0,1 levels from y=2,3 level, with the CMOS transistors (MN03,MP03). The target is the y=2,3 level. Then, it identifies if VinA is in the level y=2,3 and N2 identifies that is in the x=0,1 levels (N2=2,3), with the CMOS transistors (MN04,MP04). Signal 0: INV$_{0,7V}$(VinA) discriminates x=0 levels from y=1,2,3 level, with the CMOS transistors (MN05, MP05). The target is the y=1,2,3 level. Then, it identifies if VinA is in the level y=1,2,3 and N0 identifies that is in the x=0 levels (N0=Not 0), with the CMOS transistors (MN06,MP06). The multiplexer utilizes the selector inputs to set only one input of the selector in the level 0 or 1 or 2 or 3, that is the purpose of the negation of each signal 3,2,0 that aid to control the switches in the multiplexer. Signal 3 controls switch (MN07) to set Sucessor1=3 level (when VinA=2, SUC_r=3); signal

Table 11. SUC$_r$ transistors size

Transistor PMOS	W (μm)	L (μm)	Transistor NMOS	W (μm)	L (μm)
MP01/02	15	0.35	MN01/02	0.4	4
MP03/04	8.3	0.35	MN03/04	2.8	0.35
MP05/06	0.8	0.35	MN05/06	10	0.35
MP07 up to 11	0.4	0.35	MN07 up to 11	0.4	0.35
MP12	0.8	0.35	MN12	0.4	0.35
MP13	10	0.35	MN13/14	10	0.35

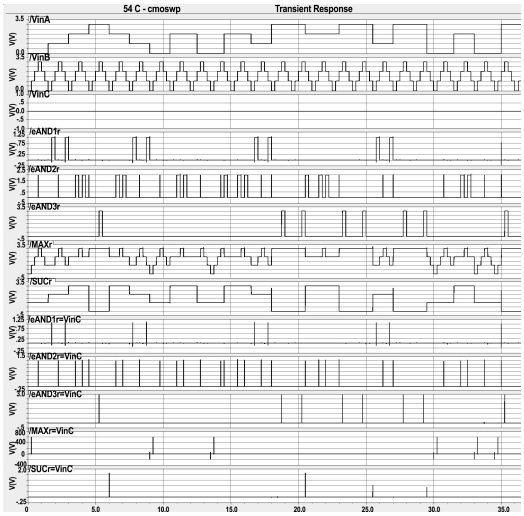

Fig. 9: Gates simulation 54°C cmostm model

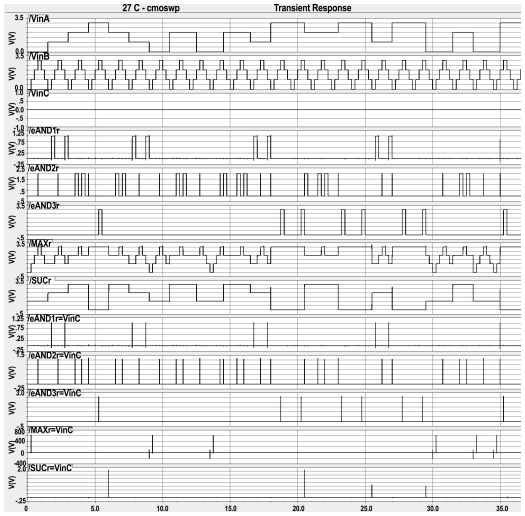

Fig. 10: Gates simulation 27°C cmoswp model

3 and N2 control switch (MP07, MP08) to set Sucessor1=2 level (when VinA=1, SUC_r=2); signal 2, N2 and N0 control switch (MN08, MN09, MP09) to set Sucessor1=1 level (when VinA=0, SUC_r=1); signal 0 controls switch (MN10) to set Sucessor1=0 level (when VinA=3, SUC_r=0). Table 11 shows the size of the CMOS transistors.

For implementation purposes of the quaternary gates to decrease circuit transistors counting it is better to implement specific circuits when Successor gates are needed in cascade instead of utilizing four, three or two SUC$_r$ gates in cascade, in which the only modification to build them is to set the correct output voltage in the original SUC$_r$ gate, this is not shown here.

4. RESULTS AND DISCUSSION

For illustration purposes, the forward and reverse computation for the $eAND_{ir}$, according to Figure 1, is shown in Figure 8. From top to bottom the correct results of the reversible gates VinA, VinB, VinC, the forward computation of $eAND_{ir}$, g0=VinA, g1=VinB (garbage), VinB=g$_{Br}$ (garbage), and the inverse computation of $eAND_{ir}$, that is $eAND_{ir}$=VinC (control signal), g0$_r$=g$_{Ar}$=VinA, g1$_r$=g$_{Br}$=VinB with the correct results are shown. Note that the input/output is a bijection, when the direct transformation is performed, the inputs are VinC=0, VinA, VinB and the outputs are $eAND_{ir}$, g$_{Br}$=VinB, g$_{Ar}$=VinA and for the inverse transformation $eAND_{ir}$=0, VinB=g$_{Br}$, VinA=g$_{Ar}$ recovering exactly the inputs.

Extensive simulations were performed for the tm,wp,ws models of the Cadence tool for 0°C, 27°C, 54°C and, for illustration purposes, Figure 9 shows the simulation for all the quaternary gates backward and forward computation for the tm model at 54°C and Figure 10 shows the simulation for all the quaternary gates backward and forward computation for the wp model at 27°C. The extensive simulations show that for all models up to 54°C all the simulations performs correctly.

The restriction for the ancillary input C=0 (only) suffices to implement the bijection in the restricted domain. Of course, it is possible to set VinC=1 or 2 or 3 to define other operator in the output, but it would increase a lot the circuit complexity without much more advantages due to the fact that, likely, each gate will be utilized to perform one operator only and if you include in one gate more operators, likely, it would be utilized only one operator and the others not, wasting a lot of operators in each gate. The implementation has the same drawbacks as the other implementation [30], as less noise rejection in comparison to the binary counterparts, increased number of transistor but less chip area, due to the fact that the interconnections are decreased which is about the 70% of the integrated circuit area.

5. CONCLUDING REMARKS AND FUTURE WORK

Reversible voltage mode quaternary gates have been implemented and verified by simulations in AMS CMOS 4ML C35B4E3 technology with results demonstrating correct functionality and feasibility of the electronic circuit. The objective is to decrease heat dissipation keeping constant the information entropy between the input and the output by means of reversible gates. Quaternary circuits have less noise rejection in comparison to the binary counterparts, increased number of transistor but less chip area. Simulations with Cadence models tm, wp, ws for temperatures: 0°C, 27°C, 54°C have shown correct functional behavior.

The universal quaternary set of gates already presented in the literature is implemented here with only 25, 18, 32, 10 and 32 CMOS transistors, respectively, outperforming past quaternary gates implementation for the non-reversible implementation by utilizing fewer CMOS transistors in more than 40%. Future works are related to further area reduction, computational performance and the IC nanometers manufacturing.

6. ACKNOWLEDGEMENT

The authors would like to thank the Technology Center of Electronics and Informatics of Mato Grosso do Sul (CTEI-MS), Federal University of Mato Grosso do Sul (UFMS) and the Conselho Nacional de Pesquisa (CNPq) for the financial support.

7. REFERENCES

[1] Feynman, R. P.: “Quantum mechanical computers”, Foundations of Physics, 1986, 16, (6), pp. 507-531.

[2] Biswas, P. K., Bahar, A. N., Habib, M. A., et al.: "Efficient design of Feynman and Toffoli gate in quantum dot cellular automata (QCA) with Energy Dissipation Analysis", Nanoscience and Nanotechnology, 2017, 7, (2), pp. 27-33.

[3] Picton, P.: "Multi-valued sequential logic design using Fredkin gates", Multiple-Valued Logic Journal, 1996, 1, (4), pp. 241-251.

[4] Thapliyal, H., Srinivas, M. B., Zwolisnki, M.: "A beginning in the reversible logic synthesis of sequential circuits". Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD), 2005, 4, pp. 6-9.

[5] Thapliyal, H., Ranganathan, N.: "Reversible logic based concurrent error detection methodology for emerging nanocircuits". 10th IEEE International Conference on Nanotechnology, Seoul, South Korea, August 2010, pp. 217-222.

[6] Morrison, M., Ranganathan, N.: "Design of a Moore finite state machine using a novel reversible logic gate, decoder and synchronous up counter". 11th IEEE International Conference on Nanotechnology, Portland, USA, August 2011, pp. 1445-1449.

[7] Taylor, M. B.: "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse". Design Automation Conference (DAC), San Francisco, USA, June 2012, pp. 1131-1136.

[8] Toffoli, T.: "Reversible computing". International Colloquium on Automata, Languages, and Programming, Berlin, Heidelberg, Germany, July 1980, pp. 632-644.

[9] Bennett, C. H.: "Logical reversibility of computation", IBM journal of Research and Development, 1973, 17, (6), pp. 525-532.

[10] Rangaraju H. G., Aakash S. Muralidhara N.: (2012). "Design and Optimization of Reversible Multiplier Circuit", International Journal of Computer Applications, 2012, (52), pp. 44-50.

[11] Singh V., Gupta R.: "A Novel n-bit Arithmetic Logic Unit Design based on Reversible Logic". IJCA Proceedings on National Symposium on Modern Information and Communication Technologies for Digital India MICTDI, 2016, pp. 27-30.

[12] A. Sadat Md. Sayem and Ueda M.: "Optimization of Reversible Sequential Circuits", Journal of Computing, 2010, 2, (6), pp.208-214.

[13] Hari, S. K. S., Shroff, S., Mahammad, S. N., et al.: "Efficient building blocks for reversible sequential circuit design". 2006 49th IEEE International Midwest Symposium on Circuits and Systems, San Juan, Puerto Rico, August 2006, pp. 437-441.

[14] Taha, S. M. R.: "Reversible logic synthesis methodologies with application to quantum computing" (Springer International Publishing, 1st edn. 2016).

[15] Athas, W. C., Svensson, L. J.: "Reversible logic issues in adiabatic CMOS". Proceedings Workshop on Physics and Computation. PhysComp'94, Dallas, USA, November 1994, pp. 111-118.

[16] Maslov, D., Dueck, G. W.: "Reversible cascades with minimal garbage", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2004, 23, (11), pp. 1497-1509.

[17] Gupta, Y. and Sasamal, T. N.: "Implementation of reversible logic gates using adiabatic logic". 2015 IEEE Power, Communication and Information Technology Conference (PCITC), Bhubaneswar, India, October 2005, pp. 595-598.

[18] Schaeffer, B., Tran, L., Gronquist, A., et al.: "Synthesis of Reversible Circuits Based on Products of Exclusive OR Sums". 2013 IEEE 43rd International Symposium on Multiple-Valued Logic, Toyama, Japan, May 2013, pp. 35-40.

[19] Singla, P., Prasad, R. R.: "Transistor Implementation Of Reversible Gate Using Novel 3 Transistor EX-OR Gate". Global Journal of Advanced Research, Jan. 2015, 2, (1), pp. 46-49.

[20] Raj, K. P., Syamala, Y.: "Transistor level implementation of digital reversible circuits", International Journal of VLSI Design & Communication Systems, 2014, 5, (6), pp. 43.

[21] Majumdar, R., Saini, S.: "A novel design of reversible 2:4 decoder". 2015 International Conference on Signal Processing and Communication (ICSC), Noida, India, March 2015, pp. 324-327.

[22] Bhardwaj, R.: "Reversible logic gates and its performances". 2018 2nd International Conference on Inventive Systems and Control (ICISC), Coimbatore, India, January 2018, pp. 226-231.

[23] Kerntopf, P., Perkowski, M., Podlaski, K.: "Synthesis of reversible circuits: A view on the state-of-the-art". 2012 12th IEEE International Conference on Nanotechnology (IEEE-NANO), Birmingham, UK, August 2012, pp. 1-6.

[24] Post, E. L.: "Introduction to a general theory of elementary propositions", American Journal of Mathematics, 1920, 43, (3), pp. 163-185.

[25] Lukasiewicz J.: "On three valued-logic.", eds. L. Borkowski (Select Works, North-Holland, Amsterdam), 1920, pp. 169-171.

[26] Hurst, S. L.: "Multiple-valued logic its status and its future", IEEE transactions on Computers, 1984, (12), pp. 1160-1179.

[27] Smith, K.: "A multiple valued logic: a tutorial and appreciation", Computer, 1988, 21, (4), pp. 17-27.

[28] KS, V. P., Gurumurthy K. S.: "Quaternary CMOS combinational logic circuits". 2009 International Conference on Information and Multimedia Technology, Jeju Islan, South Korea, December 2009, pp. 538-542.

[29] Romero, M. E. R., Martins, E. M., Santos, R. R.: "Multiple valued logic algebra for the synthesis of digital circuits". 2009 39th International Symposium on Multiple-Valued Logic, Naha, Japan, May 2009, pp. 262-267.

[30] Romero, M. E. R., Martins, E. M., Santos, R. R., Duarte, G. M.: "Universal set of CMOS gates for the synthesis of multiple valued logic digital circuits", IEEE Transactions on Circuits and Systems I: Regular Papers, 2013, 61, (3), pp. 736-749.

[31] Romero, M. E. R., Martins, E. M., Santos, R. R., Duarte, G. M.: "Analog to digital converter for binary and multiple-valued logic". 2011 IEEE Second Latin American Symposium on Circuits and Systems (LASCAS), Bogota, Colombia, February 2011, pp. 1-4.

[32] Madhuri, B. D., Sunithamani, S.: "Design of ternary logic gates and circuits using GNRFETs", IET Circuits, Devices & Systems, 2020, 14, (7), pp. 972-979.

[33] Landauer, R.: "Irreversibility and heat generation in the computing process", IBM J. Research and Development, 1961, 5, (3), pp. 183-191.

Anexo I. Código em linguagem C para minimização.

O código a seguir calcula os implicantes do cubo de base (6 variáveis, 64 mintermos), com comentários em cada trecho de código detalhando a lógica aplicada.

```
#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <limits.h>

#include <stddef.h>

#include <ctype.h>


#define MAX 4


// Essa struct representa cada célula do mapa

typedef struct{

   int binValue; // O valor da célula (0 ou 1 para o domínio binário, extensível para quaternário)

   int valueX; // O valor da numeração no eixo X

   int valueY; // O valor da numeração no eixo Y

   int valueZ; // O valor da numeração no eixo Z

   int maxValueX; // O valor de max no eixo X (número de 1s consecutivos)

   int maxValueY; // O valor de max no eixo Y (número de 1s consecutivos)

   int maxValueZ; // O valor de max no eixo Z (número de 1s consecutivos)

   int maxRangeX; // O tamanho do implicante via eixo X

   int maxRangeY; // O tamanho do implicante via eixo Y

   int maxRangeZ; // O tamanho do implicante via eixo Z

   int coverage; // Usado para replicar binValue. Um deles deve ser mutável, com atualização dos valores, o outro não.

} matrix;

// Essa struct representa um implicante. Um implicante é definido por, contando a partir de seu mintermo inferior direito,
```

```
// por posição nos eixos X, Y e Z, e tamanho(ou range) nos respectivos eixos, que representa a quantidade de 1s consecutivos
// contidos naquele implicante.
typedef struct{
   int posX;
   int posY;
   int posZ;
   int tamX;
   int tamY;
   int tamZ;
} cell;
int toGray(int n);
int posX(int n);
int posY(int n);
int posZ(int n);
unsigned hamdist(unsigned x, unsigned y);
void int_to_bin_digit(unsigned int in, int len_digit,int* out_digit);

int main(){
   int axisX, axisY, axisZ, numberOfVariables;
   int i, j, k, auxCont, c, valueX = 1, valueY = 1, valueZ = 1, maxValueX = 0, maxValueY = 0, maxValueZ = 0, maxValue = -1, flag = 0;
   int input[11] = {0, 1, 3, 5, 7, 8, 9, 10, 11, 14, 15};
   matrix mintermCell[MAX][MAX][MAX];
   cell implicant[81];

   // Seta valor 0 para tudo
   for(k = 0; k < 4; k++){
      for(j = 0; j < 4; j++){
         for(i = 0; i < 4; i++){
            mintermCell[i][j][k].binValue = 0;
            mintermCell[i][j][k].maxValueX = 0;
```

```
            mintermCell[i][j][k].maxValueY = 0;
            mintermCell[i][j][k].maxValueZ = 0;
            mintermCell[i][j][k].maxRangeX = 0;
            mintermCell[i][j][k].maxRangeY = 0;
            mintermCell[i][j][k].maxRangeZ = 0;
            mintermCell[i][j][k].implicant = 0;
        }
    }
  }

    numberOfVariables = 6;
        // As variáveis do trecho de código abaixo representam os seguintes conceitos:
    // axisX: número de cubos ao longo do eixo X (análogo para Y e Z) -> NÃO SÃO MAIS USADAS APÓS MODIFICAÇÃO DO MÉTODO
    // numberOfVariables, numberOfMinterms e numerOfCubes são auto-explicativos
    // Por exemplo, para numberOfVariables = 6, teríamos X5 X4 X3 X2 X1 X1
        // As condições abaixo atendem para numberOfVariables >= 6  if(numberOfVariables % 6 == 0){
                axisX = pow(4, numberOfVariables / 6 - 1);
                axisY = pow(4, numberOfVariables / 6 - 1);
                axisZ = pow(4, numberOfVariables / 6 - 1);
        }
        else if(numberOfVariables % 6 == 1){
                axisX = pow(4, numberOfVariables / 6 - 1) * 2;
                axisY = pow(4, numberOfVariables / 6 - 1);
                axisZ = pow(4, numberOfVariables / 6 - 1);
        }
        else if(numberOfVariables % 6 == 2){
                axisX = pow(4, numberOfVariables / 6);
                axisY = pow(4, numberOfVariables / 6 - 1);
                axisZ = pow(4, numberOfVariables / 6 - 1);
        }
```

```
        else if(numberOfVariables % 6 == 3){
                axisX = pow(4, numberOfVariables / 6);
                axisY = pow(4, numberOfVariables / 6 - 1) * 2;
                axisZ = pow(4, numberOfVariables / 6 - 1);
        }
        else if(numberOfVariables % 6 == 4){
                axisX = pow(4, numberOfVariables / 6);
                axisY = pow(4, numberOfVariables / 6) ;
                axisZ = pow(4, numberOfVariables / 6 - 1);
        }
        else if(numberOfVariables % 6 == 5){
                axisX = pow(4, numberOfVariables / 6);
                axisY = pow(4, numberOfVariables / 6);
                axisZ = pow(4, numberOfVariables / 6 - 1) * 2;
        }
        int numberOfCubes = pow(2, numberOfVariables - 6);
        int numberOfMinterms = numberOfCubes * 64;
        axisX *= 4;
        axisY *= 4;
        axisZ *= 4;
    //printf("%d, %d, %d, %d, %d\n", axisX, axisY, axisZ, numberOfCubes,
numberOfMinterms);


// Esse for coloca os mintermos de entrada, dados no vetor input(), como elementos da matriz
tridimensional (cubo)
  // Os números do vetor input() representam as células com valor 1 (mintermos), e o que não
estiver em input() permanece com valor 0
  for(i = 0; i < 11; i++){
    mintermCell[posX(input[i])][posY(input[i])][posZ(input[i])].binValue = 1;
    mintermCell[posX(input[i])][posY(input[i])][posZ(input[i])].coverage = 1;
  }
```

```
// Esse trecho de código lida com uma quantidade acima de 1 cubo, ou seja, de 6 variáveis
    // Foi feita uma matriz, por exemplo de 4 por 16, onde cada linha representa um mapa 4x4, e cada coluna seus mintermos
    // O valor 999 indica ausência de mintermos nessa nova matriz
    int contI, contJ, cubeIndex, cube[4][16], temp;
    cubeIndex = 1;
    for(j = 0; j < 4; j++){
      for(i = 0; i < 16; i++){
        cube[j][i] = 999;
      }
    }
    for(contI = 0; contI < 35; contI++){
      temp = 16 * cubeIndex;
      contJ = 0;
      while(newInput[contI] < temp){
        cube[cubeIndex - 1][contJ] = newInput[contI];
        //printf("cube[%d][%d]: %d ", cubeIndex - 1, contJ, cube[cubeIndex - 1][contJ]);
        contJ++;
        contI++;
      }
      contI--;
      cubeIndex++;
    }
    printf("\n\n");
    for(j = 0; j < 4; j++){
      for(i = 0; i < 16; i++){
        printf("%d ", cube[j][i]);
      }
      printf("\n");
    }
```

```
//Numeração dos mintermos em X
for(k = 0; k < 4; k++){
        for(j = 0; j < 4; j++){
      maxValueX = 0;
      for(i = 0; i < 4; i++){
            if(mintermCell[i][j][k].binValue == 1){
          mintermCell[i][j][k].valueX = valueX;
          valueX++;
          maxValueX++;
          flag = 0;
          auxCont = maxValueX;
            }
            else{
          mintermCell[i][j][k].valueX = 0;
          valueX = 1;
          flag = 1;
          auxCont = maxValueX;
          maxValueX = 0;
            }
            c = 1;
            while(c <= auxCont && flag == 1){
          mintermCell[i - c][j][k].maxValueX = auxCont;
          c++;
            }
        c = 0;
            while(c < auxCont && flag == 0){
          mintermCell[i - c][j][k].maxValueX = auxCont;
          c++;
            }
            if(mintermCell[i][j][k].valueX > maxValue){
          mintermCell[i][j][k].maxValueX = mintermCell[i][j][k].valueX;
```

```
        }
      }
      valueX = 1;
    }
    valueX = 1;
}
//Numeração dos mintermos em Y
for(k = 0; k < 4; k++){
       for(i = 0; i < 4; i++){
      maxValueY = 0;
      for(j = 0; j < 4; j++){
            if(mintermCell[i][j][k].binValue == 1){
          mintermCell[i][j][k].valueY = valueY;
          valueY++;
          maxValueY++;
          flag = 0;
          auxCont = maxValueY;
            }
            else{
          mintermCell[i][j][k].valueY = 0;
          valueY = 1;
          flag = 1;
          auxCont = maxValueY;
          maxValueY = 0;
            }
        c = 1;
            while(c <= auxCont && flag == 1){
          mintermCell[i][j - c][k].maxValueY = auxCont;
          c++;
            }
        c = 0;
```

```
                while(c < auxCont && flag == 0){
            mintermCell[i][j - c][k].maxValueY = auxCont;
            c++;
                }
                if(mintermCell[i][j][k].valueY > maxValue){
            mintermCell[i][j][k].maxValueY = mintermCell[i][j][k].valueY;
          }
       }
       valueY = 1;
    }
        valueY = 1;
}
//Numeração dos mintermos em Z
for(j = 0; j < 4; j++){
         for(i = 0; i < 4; i++){
       maxValueZ = 0;
       for(k = 0; k < 4; k++){
                if(mintermCell[i][j][k].binValue == 1){
            mintermCell[i][j][k].valueZ = valueZ;
            valueZ++;
            maxValueZ++;
            flag = 0;
            auxCont = maxValueZ;
                }
                else{
            mintermCell[i][j][k].valueZ = 0;
            valueZ = 1;
            flag = 1;
            auxCont = maxValueZ;
            maxValueZ = 0;
                }
```

```
            c = 1;
                    while(c <= auxCont && flag == 1){
               mintermCell[i][j][k - c].maxValueZ = auxCont;
               c++;
                    }
            c = 0;
                    while(c < auxCont && flag == 0){
               mintermCell[i][j][k - c].maxValueZ = auxCont;
               c++;
                    }
                    if(mintermCell[i][j][k].valueZ > maxValue){
               mintermCell[i][j][k].maxValueZ = mintermCell[i][j][k].valueZ;
            }
          }
          valueZ = 1;
       }
       valueZ = 1;
    }


// Nesse trecho ocorre a extração de implicantes.
    // São feitos várias verificações (ifs), pois existem vários possíveis "critérios de parada" quando se analisa um implicante.
    // Inicia-se na última célula de cada mapa (de modo que, na caminhada do for, sempre encontra-se inicialmente o mintermo
    // com maiores valores para se verificar o implicante do qual faz parte).
    // Recomenda-se fazer um teste de mesa executando esse algoritmo para o exemplo de input visto neste código, de modo a
    // facilitar a compreensão do mesmo, entendendo a necessidade de cada if.
    int newRangeX, newRangeY, cont, newi, newj, newk, implicantCont, flagImplicant, newImplicantCont, neweri, newerj, contY;
    newRangeX = 0; newRangeY = 0; newk = 0; implicantCont = 1, flagImplicant = 0, newImplicantCont = 0;
    for(newj = 3; newj >= 0; newj--){
```

```
newi = 3;
newRangeX = mintermCell[newi][newj][newk].maxValueX;
newRangeY = mintermCell[newi][newj][newk].maxValueY;
for(newi = 3; newi >= 0; newi--){
   if(mintermCell[newi][newj][newk].binValue == 1){
      //printf("%d %d", newRangeX, newRangeY);
      if(mintermCell[newi][newj][newk].maxValueX <= newRangeX){
         newRangeX = mintermCell[newi][newj][newk].maxValueX;
      }
      if(newRangeX == 0){
         newRangeX = mintermCell[newi][newj][newk].maxValueX;
      }
      if(mintermCell[newi][newj][newk].maxValueY <= newRangeY){
         newRangeY = mintermCell[newi][newj][newk].maxValueY;
      }
      if(newRangeY == 0){
         newRangeY = mintermCell[newi][newj][newk].maxValueY;
      }
      if(newRangeX == 3){
         newRangeX = pow(2, floor(log2(newRangeX)));
      }
      if(newRangeY == 2){
         newRangeY = pow(2, floor(log2(newRangeY)));
      }
      if(newi == 0){
         for(contY = 0; contY < newRangeY; contY++){
            for(cont = 0; cont < newRangeX; cont++){
               mintermCell[newi + cont][newj + contY][newk].binValue = 9;
               flagImplicant = 1;
            }
         }
```

```
                implicant[newImplicantCont].tamX = newRangeX;
implicant[newImplicantCont].tamY = newRangeY;

                implicant[newImplicantCont].posX = newi + cont - 1;
implicant[newImplicantCont].posY = newj + contY - 1; implicant[newImplicantCont].posZ =
newk;

                newImplicantCont++;

                if(flagImplicant == 1){

                  implicantCont++;

                  flagImplicant = 0;

                }

            }

          }

          else{

            if(mintermCell[newi + 1][newj][newk].valueY < mintermCell[newi +
1][newj][newk].maxValueY){

              for(contY = 0; contY < newRangeY; contY++){

                for(cont = 0; cont < newRangeX; cont++){

                  mintermCell[newi + 1 + cont][newj + contY][newk].binValue = 9;

                  flagImplicant = 1;

                }

              }

              if(flagImplicant == 1){

                implicant[newImplicantCont].tamX = newRangeX;
implicant[newImplicantCont].tamY = newRangeY;

                implicant[newImplicantCont].posX = newi + cont;
implicant[newImplicantCont].posY = newj + contY - 1; implicant[newImplicantCont].posZ =
newk;

                newImplicantCont++;

                implicantCont++;

                flagImplicant = 0;

              }

            }

            else{

              for(contY = 0; contY < newRangeY; contY++){
```

```
                for(cont = 0; cont < newRangeX; cont++){
                    mintermCell[newi + 1 + cont][newj - contY][newk].binValue = 9;
                    flagImplicant = 1;
                }
            }
            if(flagImplicant == 1){
                implicant[newImplicantCont].tamX = newRangeX;
implicant[newImplicantCont].tamY = newRangeY;
                implicant[newImplicantCont].posX = newi + cont;
implicant[newImplicantCont].posY = newj; implicant[newImplicantCont].posZ = newk;
                newImplicantCont++;
                implicantCont++;
                flagImplicant = 0;
            }
          }
          newRangeX = 0; newRangeY = 0;
        }
      }
   }

   int testCont;
   printf("\n\nImplicantArray\n");
   for(i = 0; i < newImplicantCont; i++){
      printf("pos X: %d | pos Y: %d | pos Z: %d | tam X: %d | tam Y: %d\n", implicant[i].posX,
implicant[i].posY, implicant[i].posZ, implicant[i].tamX, implicant[i].tamY);
   }
   printf("\n");
}

// Obtém o valor Gray dentro de um mapa 4x4x4 (apenas inverte de posição os valores 11 e
10, no domínio binário)
int toGray(int n){
```

```
    if(n == 2){
        return 3;
    }
    else if(n == 3){
        return 2;
    }
}

// Mapeamento de cubos de ordem superior
int posX(int n){
    n = n % 4;
    n = toGray(n);
    return n;
}
int posY(int n){
    n = (int)floor(n / 4) % 4;
    n = toGray(n);
    return n;
}
int posZ(int n){
    n = (int)floor(n / 16) % 4;
    n = toGray(n);
    return n;
}

// Distância Hamming entre dois mintermos
unsigned hamdist(unsigned x, unsigned y)
{
  unsigned dist = 0, val = x ^ y;

  // Count the number of set bits
```

```
  while(val)
  {
    ++dist;
    val &= val - 1;
  }

  return dist;
}

// Converte um número decimal em um vetor binário
void int_to_bin_digit(unsigned int in, int len_digit,int* out_digit){

  unsigned int mask = 1U << (len_digit-1);
  int i;
  for (i = 0; i < len_digit; i++) {
    out_digit[i] = (in & mask) ? 1 : 0;
    in <<= 1;
  }
}
```