UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE TELEMÁTICA

CONTRIBUIÇÃO À MINIMIZAÇÃO E SIMULAÇÃO DE CIRCUITOS LÓGICOS

Autor: Alexandre César Rodrigues da Silva

Membros da Banca:

Presidente: Prof. Dr. Ivanil Sebastião Bonatti
Membro: Dr. Mário L. Côrtes
Membro: Prof. Dra. Beatriz Máscia Daltrini
Suplente: Prof. Dr. Walter da Cunha Borelli

Este exemplar corresponde à redação final da tese defendida por Alexandre César Rodrigues da Silva e aprovada pela comissão julgadora em 10/11/89 Bonatti

Tese apresentada à Faculdade de Engenharia Elétrica como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica.

Novembro de 1989

AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos a todos os amigos, professores e funcionários da FEE que direta ou indiretamentamente contribuíram para a elaboração desta tese, e em especial, ao Prof. Ivanil Sebastião Bonatti pela sua orientação e contínuo incentivo e à sua família pela amizade demonstrada.

Aos doutores Mário L. Côrtes, Beatriz Máscia Daltrini e Walter da Cunha Borelli gostaria de agradecer a gentileza da aceitação de comporem a banca examinadora desta tese.

Aos colegas Madureira, Giordano, Gorgônio e Miguel gostaria de agradecer a colaboração direta nos trabalho desta tese.

Gostaria de agradecer, em particular, à minha família pelo apoio que recebi durante todo este trabalho.

Este trabalho de pesquisa contou com o apoio da Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES) e do Departamento de Telemática da FEE - UNICAMP.

RESUMO

Este trabalho é relacionado à síntese, à análise e à simplificação de circuitos lógicos.

A álgebra booleana e as técnicas de detecção de falhas são apresentadas como introdução ao estudo dos circuitos lógicos.

Um algoritmo para cobertura irredundante de funções booleanas é apresentado. Ele é baseado num método originalmente desenvolvido para análise de falhas.

Comparações realizadas com o algoritmo de Quine-McCluskey e com o algoritmo de Caruso mostraram que o método apresentado tem um desempenho melhor que estes dois quanto ao uso de memória.

As máquinas sequenciais foram apresentadas junto com um procedimento para redução de estados e com um programa que sintetiza circuitos lógicos a partir dos diagramas de estados destas máquinas.

Uma versão melhorada do programa LóGICO é apresentada e seu desempenho é ilustrado através dos resultados de seu uso em alguns circuitos lógicos práticos.

Abstract

This work deals with some aspects related to synthesis, analysis and simplification of logic circuits.

The boolean algebra is introduced through basic axioms, as well as the d-algorithm for fault detection as a indispensable mathematical tool for studying logical circuits.

For the minimization of boolean functions a procedure that yields a quasi-minimum cover to the functions is presented. It is based on algorithms originally developped for failure diagnosis.

Comparisons are made with an algebraic procedure based on the Quine-McCluskey method and with an improved version of Caruso's method. Numerical studies have shown that the presented method performs better than the ones cited above with regard to workspace requirements.

The sequential machines are presented along with a procedure for state reduction and a program that realizes the logic circuits from their Mealy's state diagrams.

An improved version of the LOGICO program is presented and used in some cases of practical circuits.

ÍNDICE

C A P Í T U L O I

INTRODUÇÃO

I.1.- INTRODUÇÃO

O barateamento com consequente popularização dos componentes digitais tornou realidade e difundiu a expressão "Projeto Auxiliado por Computador".

Assim, a tarefa do engenheiro projetista é hoje cada vez mais eficiente, pois conta com um número crescente de "software" de síntese e análise nas mais variadas atividades técnicas.

O projeto de equipamentos digitais consiste essencialmente das seguintes etapas:

A.- Descrição funcional, entrada-saída, do circuito. Em geral isto é feito através de diagrama de tempo e diagrama de estado;

B.- Partição preliminar do circuito em blocos, com definição das interfaces entre os blocos;

C.- Descrição formal, entrada/saída de cada bloco;

D.- Síntese de cada bloco com minimização das funções booleanas;

E.- Análise de desempenho;

F.- Reavaliação do projeto com base no seu desempenho, podendo-se retornar ao passo A) ou seguir ao passo G);

G.- Implementação final.

As etapas C), D) e E) são as que mais se prestam a automatização e são nestas áreas justamente que se concentra este trabalho de tese. A figura I.1 apresenta o esquema acima descrito de projeto de circuitos digitais.

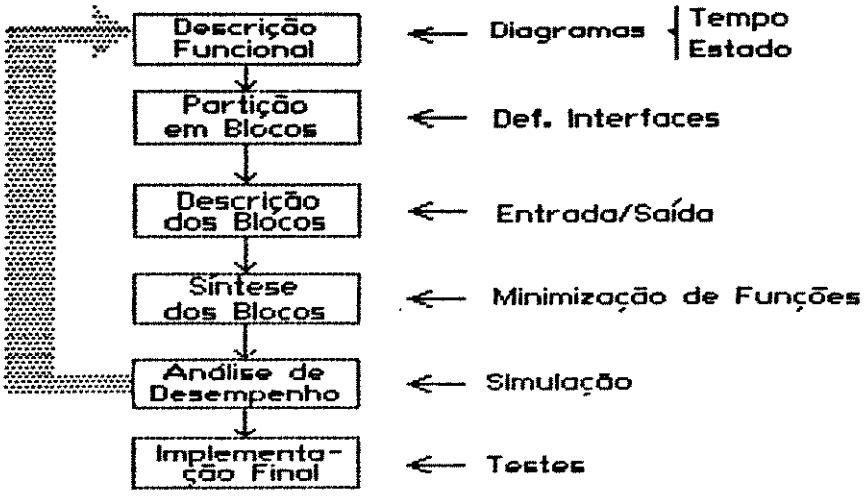


FIGURA I.1- Esquema de projeto de circuitos digitais.

Este trabalho é relacionado à síntese, à análise e à simplificação de circuitos lógicos.

No capítulo II definem-se as funções booleanas afim de se descrever as propriedades dos circuitos combinacionais. Através de postulados básicos apresenta-se a álgebra de Boole, ferramenta matemática essencial ao estudo dos circuitos lógicos. Foi dada uma ênfase particular à geração de testes para falhas em circuitos combinacionais, com intuito de introduzir os conceitos básicos utilizados no método de Cobertura Irredundante de Funções Booleanas apresentado no capítulo III.

O capítulo III trata da minimização de funções booleanas, onde são descritos três algoritmos: o algoritmo de Quine-McCluskey; o algoritmo de Caruso e o algoritmo de "Cobertura Irredundante Através da Geração de Padrão de Teste" desenvolvido neste trabalho.

O problema de simplificar funções booleanas consiste em obter a partir da tabela verdade uma expressão booleana mínima de acordo com algum critério de custo.

No capítulo IV são tratados os circuitos sequenciais.

Com o intuito de sistematizar o projeto de máquinas seguenciais síncronas é construida uma tabela de estados a partir de seu diagrama de estado. Esta tabela pode, frequentemente, ser reduzida através de uma análise de equivalência de estados.

A sistematização e automação de projetos de circuitos digitais é extremamente conveniente pois retira do projetista as tarefas cansativas, sujeitas a erros e demoradas, para permitir-lhe uma análise crítica das

possíveis soluções. Neste capítulo é apresentada uma versão melhorada do programa TABELA e sua utilidade é ilustrada através de alguns "projetos".

O resultado do programa TABELA é um circuito que corresponde a uma realização da máquina de estado. A validação de um projeto passa necessariamente por sua simulação. O programa LóGICO desenvolvido inicialmente por Madureira [1] realiza a simulação de circuitos lógicos sequenciais e combinacionais. No capítulo V uma versão melhorada do programa LóGICO é apresentada e seu desempenho é ilustrado através do seu uso em alguns circuitos lógicos práticos.

I.2.- REFERÊNCIAS BIBLIOGRÁFICAS

[1] M. C. G. Madureira
Contribuição à Análise e Síntese de Circuitos Digitais
Tese de Mestrado - FEE - UNICAMP, Maio de 1987

C A P Í T U L O II

CIRCUITOS COMBINACIONAIS

II.1.- INTRODUÇÃO

Neste capítulo definem-se as funções booleanas com a finalidade de descrever as propriedades dos circuitos combinacionais. A característica particular de um circuito combinacional é que suas saídas são funções unicamente de suas entradas presentes.

Através de postulados básicos apresenta-se a álgebra de Boole como ferramenta matemática essencial ao estudo de circuitos combinacionais.

Uma ênfase particular é dada na definição de testes para falhas em circutos combinacionais.

São introduzidos conceitos e notações utilizados em todo o trabalho.

II.2.- ÁLGEBRA DE BOOLE

Os conceitos básicos da álgebra de Boole são introduzidos como um conjunto de postulados dos quais derivam teoremas importantes, que se constituem nas ferramentas necessárias para a manipulação e simplificação das expressões algébricas booleanas.

II.2.1.- POSTULADOS FUNDAMENTAIS

O postulado básico da álgebra de Boole é a existência de uma variável booleana tal que:

$$x \neq 0 \iff x = 1$$
$$x \neq 1 \iff x = 0$$

A álgebra de Boole é um sistema algébrico que consiste do conjunto (0,1),

duas operações binárias chamadas OR (+) e AND (.) e uma operação unária chamada NOT denotada por um apóstrofo (') sobre a variável.

A operação OR é chamada de soma lógica ou união, a operação AND é conhecida por produto lógico ou interseção e a operação NOT é dita complementação. Estas operações são definidas conforme a tabela II.1 e podem ser representadas por circuitos elementares denominados gates ou portas, mostrados na figura II.1.

OR	AND	NOT
0 + 0 = 0	0 . 0 = 0	0' = 1
0 + 1 = 1	0 . 1 = 0	1' = 0
1 + 0 = 1	1 . 0 = 0	
1 + 1 = 1	1 . 1 = 1	

Tabela II.1.- Definição das operações OR, AND e NOT.

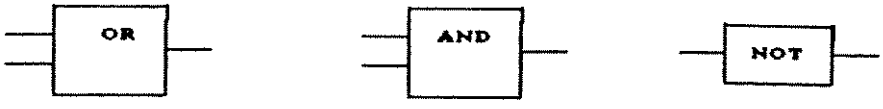


Figura II.1.- Símbolos das portas lógicas OR, AND e NOT.

Um circuito composto de gates é denominado de circuito lógico. A figura II.2 ilustra um exemplo de um circuito que realiza a expressão lógica:

$$A.B + C + D.E$$

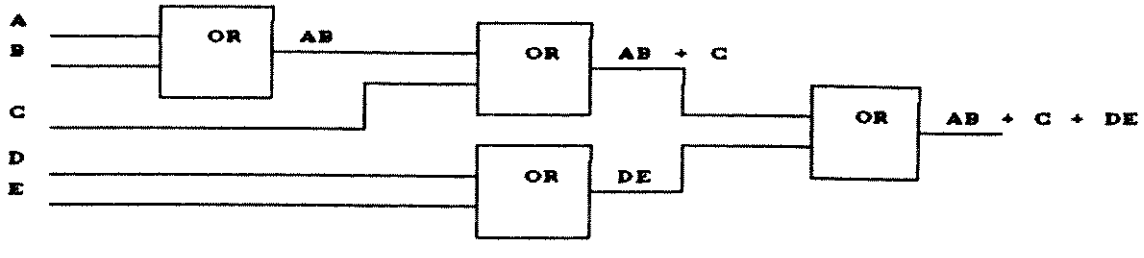


Figura II.2.- Exemplo de Circuito Lógico.

II.2.2.- PROPRIEDADES BÁSICAS

Se x é uma variável booleana, então :

$$x + 1 = 1 \quad \text{(II.1)}$$

$$x \cdot 0 = 0 \quad \text{(II.2)}$$

$$x + 0 = x \quad \text{(II.3)}$$

$$x \cdot 1 = x \quad \text{(II.4)}$$

$$x + x = x \quad \text{(II.5)}$$

$$x \cdot x = x \quad \text{(II.6)}$$

A álgebra booleana também é comutativa e associativa em relação às duas operações binárias. Os parênteses são colocados da mesma forma que na álgebra convencional, ou seja, x+y.z é x+(y.z) e não (x+y).z . Sendo x, y, z variáveis booleanas, então

-Comutativa

$$x + y = y + x \quad \text{(II.7)}$$

$$x \cdot y = y \cdot x \quad \text{(II.8)}$$

-Associativa

$$(x + y) + z = x + (y + z) \quad \text{(II.9)}$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \text{(II.10)}$$

-Complemento

$$x + x' = 1 \quad \text{(II.11)}$$

$$x \cdot x' = 0 \quad \text{(II.12)}$$

Na álgebra booleana, a soma é distributiva sobre o produto e o produto é distributivo sobre a soma,

-Distributiva

$$x \cdot (y + z) = x.y + x.z \quad \text{(II.13)}$$

$$x + (y \cdot z) = (x + y) \cdot (x + z) \quad \text{(II.14)}$$

Todas estas propriedades podem ser provadas por indução perfeita. Observa-se ainda que todas elas se apresentam aos pares e que, em cada par, uma equação pode ser obtida da outra trocando-se as constantes 0 por 1 e 1 por 0, e as operações AND por OR e OR por AND. Isto é conhecido como princípio da dualidade da álgebra de Boole. A implicação disto é que basta provar apenas uma das equações e sua dual estará provada.

II.2.3.- EXPRESSÕES BOOLEANAS

Define-se expressão booleana como a combinação de um número finito de variáveis booleanas e constantes (0,1) através de operações booleanas (+), (.), (').

Qualquer constante ou variável booleana é uma expressão booleana, e se T1 e T2 são expressões booleanas, também o são T1',T2', T1+T2, T1.T2.

As propriedades (II.15) a (II.20) a seguir formam o conjunto de ferramentas básicas para a simplificação de expressões booleanas. Elas estabelecem a noção de redundância e aparecem em pares duais :

- Absorção

$$x + xy = x \qquad \text{(II.15)}$$

$$x(x + y) = x \qquad \text{(II.16)}$$

$$x + x'y = x + y \qquad \text{(II.17)}$$

$$x(x' + y) = xy \qquad \text{(II.18)}$$

- Consenso

$$xy + x'z + yz = xy + x'z \qquad \text{(II.19)}$$

$$(x + y)(x' + z)(y + z) = (x + y)(x' + z) \qquad \text{(II.20)}$$

OBS.: Sempre que não deixar dúvidas, a forma xy é utilizada no lugar de x.y.

As propriedades II.1 a II.20 permitem uma variedade imensa de manipulações nas expressões booleanas. Sempre que possível, elas permitem converter uma expressão em outra equivalente, com menos literais, onde literal é o aparecimento de uma variável ou de seu complemento. Por exemplo, o lado esquerdo da equação (II.19) tem 6 literais, enquanto o lado direito tem apenas 4. Se o valor de uma expressão independe do valor de uma variável x, x é dito redundante. Assim, estas propriedades podem eliminar literais redundantes numa dada expressão, simplificando-a.

É importante ressaltar que não são definidas operações inversas na álgebra de Boole e, consequentemente, não são permitidos cancelamentos. Por exemplo, se A + B = A + C, não está implícito que B = C. De fato, se A = B = 1 e C = 0, 1+1 = 1+0, embora B seja diferente de C. Analogamente, AB = AC não implica em B = C.

Teorema de DE MORGAN

As regras que regem a operação de complementação se resumem nos três teoremas seguintes:

$$(x')' = x \qquad \text{(II.21)}$$

$$(x + y)' = x'.y' \qquad \text{(II.22)}$$

$$(x.y)' = x' + y' \qquad \text{(II.23)}$$

As equações II.22 e II.23 são conhecidas como "teoremas de De Morgan" para duas variáveis. De forma geral, este teorema afirma que o complemento de qualquer expressão pode ser obtido trocando-se simultaneamente cada variável por seu complemento e as operações OR por AND e AND por OR, ou seja:

$$[f(x1,x2,\ldots,xn,0,1,+,.)]' = f(x1',x2',\ldots,xn',1,0,.,+) \qquad (II.24)$$

II.3.- FUNÇÕES BOOLEANAS

Seja $T(x1,x2,\ldots,xn)$ uma expressão booleana. Como cada uma das n variáveis x1...xn pode independentemente assumir um dos dois valores 0 ou 1, existem 2^n combinações de valores a serem considerados na determinação dos valores de T. Para determinar o valor de uma expressão para uma dada combinação de valores, basta substituir os valores das variáveis na expressão. Por exemplo, para

$$T(x,y,z) = x'z + xz' + x'y'$$

se x=0, y=0 e z=1, temos :

$$T(0,0,1) = 0'.1 + 0.1' + 0'.0'$$
$$= 1.1 + 0.0 + 1.1$$
$$= 1 + 0 + 1 = 1$$

Da mesma forma, os valores de T podem ser calculados para todas as combinações possíveis de x, y e z como na tabela II.2.

x	y	z	T
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Tabela II.2.- Tabela verdade para $T(x,y,z) = x'z + xz' + x'y'$.

Se este procedimento for repetido para se construir a tabela verdade da expressão $x'z + xz' + y'z'$, será obtida uma tabela idêntica à tabela II.2. Isto é, expressões booleanas diferentes podem ser representadas pela mesma tabela verdade. Os valores assumidos por uma expressão para todas as combinações possíveis de suas variáveis definem uma função booleana.

Em outras palavras, uma função booleana f(x1,...,xn) é a correspondência que associa um elemento da álgebra de Boole com cada uma das 2^n combinações das variáveis x1,...,xn. Uma boa maneira de expressar tal correspondência é através de uma tabela verdade. Vale observar que cada tabela verdade define apenas uma função, embora esta função possa ser expressa por diferentes expressões booleanas.

II.3.1.- FORMAS CANÔNICAS

Na seção anterior, as funções booleanas foram representadas através de tabelas verdade. Uma expressão que represente uma função booleana é derivada de sua tabela encontrando a soma de todos os termos para os quais a função assume o valor 1. Cada termo é um produto de variáveis da função. Uma variável xi aparecerá neste produto na forma complementada se o seu valor for 0 nesta combinação, e na forma não complementada se seu valor for 1.

Por exemplo, a função definida na tabela II.2 assume valor 1 para a combinação x=y=z=0 (linha 1) assim, o termo x'y'z' aparece na fórmula da função. De fato, para a tabela II.2, pode-se definir :

$$f(x,y,z) = x'y'z' + x'y'z + x'yz + xy'z' + xyz'.$$

Um produto que contém todas as n variáveis de uma função como fatores, complementadas ou não, é chamado de mintermo. Sua característica particular é que assume o valor 1 para somente uma combinação de valores das variáveis. A soma de todos os mintermos derivados das linhas da tabela onde a função assume o valor 1, assumirá os valores 0 ou 1 da mesma forma que a função. Portanto, esta soma é realmente uma representação algébrica da função. Uma expressão deste tipo é chamada de soma canônica de produtos.

REPRESENTAÇÃO DECIMAL DE COMBINAÇÕES BINÁRIAS:

Um número decimal inteiro positivo é representado na forma Dn...D2D1D0, que corresponde ao valor

$$D0 + D1*10^1 + D2*10^2 + \ldots + Dn*10^n$$

onde D0, D1,..., Dn são dígitos que só podem assumir os valores 0, 1, 2,...,9.Desta forma as n-uplas binárias correspondentes a cada uma das combinações dos possíveis valores das variáveis booleanas podem ser interpretadas como um número binário inteiro positivo.

Assim, a n-upla Bn...B2B1B0 corresponde ao valor decimal

$$B0 + B1*2^1 + B2*2^2 + \ldots + Bn*2^n$$ onde Bn, ..., B2, B1, B0

só podem assumir os valores 0 e 1.

Exemplos:

$$(101)_2 = 1 + 0*2^1 + 1*2^2 = 1 + 4 = 5$$
$$(110)_2 = 0 + 1*2^1 + 1*2^2 = 2 + 4 = 6$$

As funções booleanas são normalmente expressas numa forma compacta, obtida pelos códigos decimais associados aos mintermos para os quais a função vale 1 . Estes decimais são derivados das tabelas verdades tratando-se cada linha como um número binário, por exemplo o mintermo x'yz é associado à linha 011 que, interpretada como um número binário, é igual ao decimal 3. Assim, a função definida pela tabela II.2 pode ser expressa como f(x,y,z) = S(0,1,3,4,6) onde S( ) significa que f(x,y,z) é a soma de todos os mintermos cujos códigos decimais são dados entre os parênteses.

Uma função booleana também pode ser expressa por um produto de somas. Este pode ser obtido considerando-se os termos para os quais a função assume o valor 0. Cada termo é uma soma de variáveis da função. Uma variável xi aparecerá nesta soma na forma complementada se o seu valor for 1 nesta combinação, e na forma não complementada se seu valor for 0.

Por exemplo, a função definida na tabela II.2 assume valor 0 para a combinação x=y=z=1 (linha 7); assim, o termo x'+y'+z' aparece na fórmula da função. De fato, para a tabela II.2, pode-se definir

$$f(x,y,z) = (x+y'+z).(x'+y+z').(x'+y'+z')$$

Uma soma que contém todas as n variáveis de uma função como fatores, complementadas ou não, é chamada de maxtermo. O produto de todos os maxtermos derivados das linhas da tabela onde a função assume o valor 0, assumirá os valores 0 ou 1 da mesma forma que a função. Portanto, este produto também é uma representação algébrica da função e é chamado de produto canônico de somas. Também como produto de somas, as funções booleanas podem ser expressas numa forma compacta, obtida pelos códigos decimais associados aos maxtermos para os quais a função vale 0.

Assim, a função definida pela tabela II.2 pode ser expressa também na forma

$$f(x,y,z) = P(2,5,7)$$

onde P( ) significa que f(x,y,z) é o produto de todos os maxtermos cujos códigos decimais são dados entre os parênteses.

Existem situações onde, enquanto a função assume o valor 1 para algumas combinações de entradas e 0 para outras, pode assumir qualquer valor para algumas combinações de entrada. Estas situações podem ocorrer, por exemplo, quando as variáveis não são mutuamente independentes, ou seja, uma dependência

entre as variáveis pode impedir a ocorrência de certas combinações, para as quais o valor da função não é especificado. Estas combinações das entradas são chamadas de "Don't Care States" ou Estados Irrelevantes. O valor da função para tais combinações é denotado por um X ou um traço (-) e estas combinações aparecem nas formas compactas como D( ).

Por exemplo :

$$F(w,x,y,z) = S(4,5,8,12,13) + D(3,14,15)$$

w	x	y	z	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	X
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	X

TABELA II.3.- Tabela verdade para F(wxyz)=P(4,5,8,12,13) + D(3,14,15)

II.3.2.- CUSTOS

Definidas as formas de representação das funções, e tendo em vista a simplificação das expressões através dos teoremas definidos na seção II.2.1, cabe agora definir que direção seguir quando se fala em simplificação. O objetivo do projetista de circuitos digitais vai no sentido de "baratear" o produto, garantindo sua qualidade. Assim, deve-se deixar claro o critério de custo considerado.

Alguns critérios de custo mínimo de uma dada função booleana são:

1.- Menor número de literais. (Lembrando que uma literal é uma variável na forma complementada ou não).

2.- Menor número de literais numa expressão do tipo soma de produtos (ou produto de somas).

3.- Menor número de termos numa expressão do tipo soma de produtos (ou produto de somas), sendo que não exista nenhuma outra expressão com o mesmo número de termos e menor número de literais.

Neste trabalho é adotado o critério 3, e para as expressões mínimas das funções é adotada a forma de soma de produtos.

Algumas vezes, porém, a forma de produto de somas pode ser mais vantajosa e a função deve ser minimizada a partir de seus maxtermos. Entretanto, de acordo com este critério de custos e por aplicação do teorema de De Morgan (II.24), o custo obtido a partir dos maxtermos é equivalente ao custo obtido a partir dos mintermos da função negada.

EXEMPLO: Retomando-se a função dada pela tabela II.2, $F(x,y,z) = S(0,1,3,4,6)$

Escrevendo-a na forma de soma de produtos para uma primeira avaliação do seu custo, tem-se:

$$f(x,y,z) = x'y'z' + x'y'z + x'yz + xy'z' + xyz'$$

$$CUSTO = (3 + 3 + 3 + 3 + 3) + 5 = 20$$

Com alguma manipulação obtém-se, para a mesma função,

$$f(x,y,z) = x'y' + x'z + y'z' + xz'$$

$$CUSTO = (2 + 2 + 2 + 2) + 4 = 12$$

Um pouco mais de simplificação e chega-se a sua fórmula mínima :

$$f(x,y,z) = x'y' + x'z + xz'$$

$$CUSTO = (2 + 2 + 2) + 3 = 9$$

A função custo pode ser interpretada como equivalente a contagem dos terminais de entrada dos gates envolvidos na realização da função, sem se contar os inversores. Na forma final dessa função, pode-se observar que serão necessários 3 AND de 2 entradas e um OR de 3 entradas, isto é:

$$3x2 + 1x3 = 6+3 = 9$$

II.3.3.- MAPA DE KARNAUGH

Um mapa de Karnaugh é uma forma modificada da tabela verdade, na qual as combinações das entradas estão arranjadas de uma forma particularmente conveniente. Os mapas para funções de 2 a 5 variáveis são mostrados na figura II.3. O mapa de Karnaugh para funções de 6 variáveis está na figura II.4. Na parte externa dos mapas representam-se as variáveis e as combinações de seus

valores. Cada mapa de n variáveis consiste de 2^n células (quadrados), representando todas as possíveis combinações dessas variáveis. Os códigos decimais correspondentes a essas combinações também estão nas figuras II.3 e II.4.

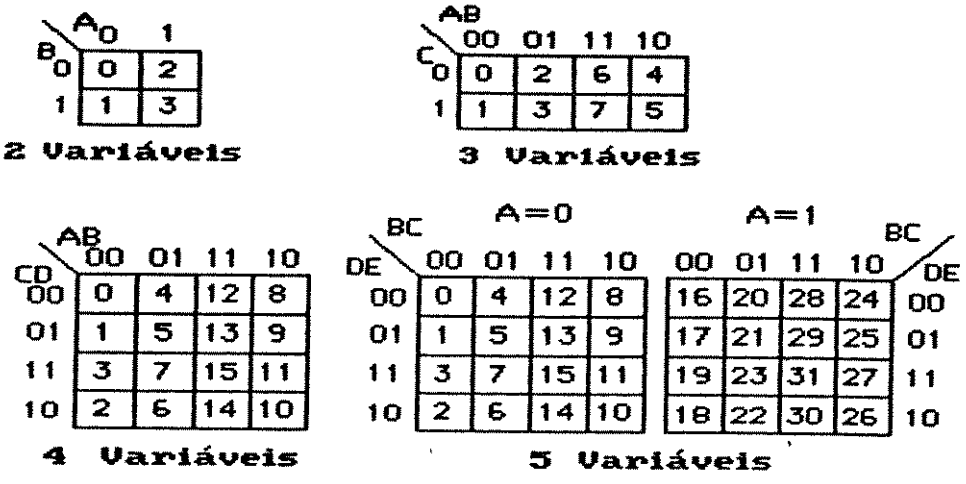


Figura II.3.- Mapa de Karnaugh para 2 a 5 variáveis

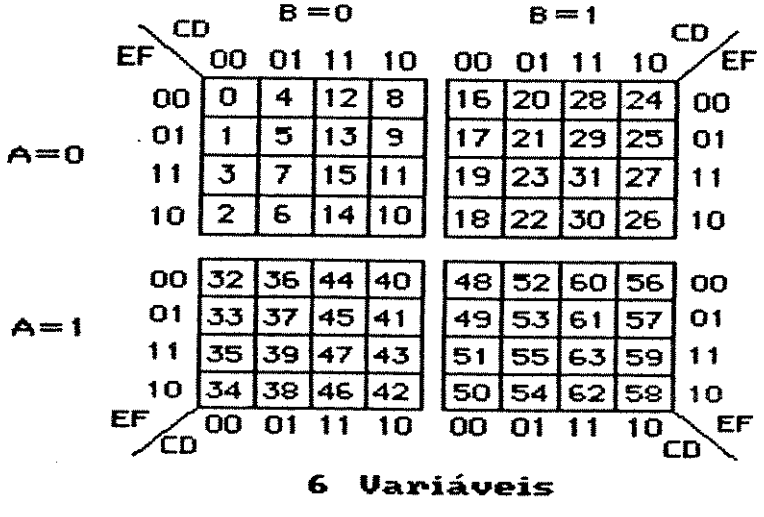


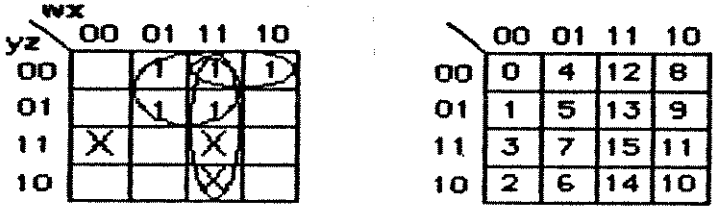
Figura II.4.- Mapa de Karnaugh para 6 variáveis.

O valor da função associado a uma combinação particular das entradas é anotado na célula correspondente. Por exemplo, a figura II.5 mostra o mapa para a função :

$$F(w,x,y,z) = S(4,5,8,12,13) + D(3,14,15)$$

Note que o valor 1 está nas células 4,5,8,12,13, o valor X nas células 3, 14 e 15. As células em branco correspondem às combinações para as quais a função vale 0. O mintermo correspondente a uma célula particular é determinado

da mesma forma que na tabela verdade. Uma variável aparece complementada no produto se tiver valor 0 na célula correspondente, e não complementada se tiver valor 1. Por exemplo, a célula 5 na figura II.5 corresponde a w'xy'z .



	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

$$F(w,x,y,z) = S(4,5,8,12,13) + D(3,14,15)$$

Figura II.5.- Representação de uma função no mapa de Karnaugh.

O código usado para identificar as colunas e linhas é extremamente importante. Graças a este código, células com um lado em comum correspondem a combinações que diferem no valor de uma única variável. Estas células são chamadas de adjacentes. Com o propósito de se determinar adjacências deve-se imaginar o mapa de 3 variáveis como a versão topológica plana de um cilindro, ou seja, as células 0 e 4 são adjacentes, assim como as células 1 e 5. Da mesma forma, o mapa de 4 variáveis deve ser imaginado como um toróide (câmara de ar), onde a célula 8 é vizinha das células 0 e 10.

Isto torna-se importante no contexto de simplificação e minimização de funções, uma vez que células adjacentes podem ser combinadas de acordo com a regra xy + xy' = x.

Assim, o produto correspondente à união de 2 células adjacentes para as quais a função assume valor 1 é obtido escrevendo-se todas as variáveis que assumem o mesmo valor nas duas células, ignorando-se aquelas que estão na forma complementada em uma célula e não complementada na outra. Por exemplo a união das células 8 e 12 na figura II.5 corresponde ao produto wy'z', pois wxy'z'+wx'y'z' = wy'z'(x+x') = wy'z'.

Um conjunto de 2^m células, cada uma adjacente a m células do conjunto é chamado de subcubo de ordem m, e diz-se que o subcubo cobre estas células. Todo subcubo cuja ocorrência ("1" lógico) obriga a função a ter valor "1", ou seja, IMPLICA a função dada é um implicante da função. A cada implicante da função corresponde um produto de literais.

Uma função pode ser expressa como a soma dos produtos (implicantes)

correspondentes aos subcubos necessários para cobrir todas as suas células de valor 1. Por exemplo, a função dada na figura II.5 pode ser expressa como a soma dos 3 implicantes correspondentes aos subcubos assinalados ,

$$F = wx + xy' + wy'z' \qquad (II.25)$$

Afim de se obter uma expressão mínima para a função, deve-se cobrir todas as células de valor 1 com o menor número de subcubos, sendo que cada subcubo seja tão grande quanto possível, pois o número de termos na expressão depende do número de subcubos, enquanto o número de literais é menor à medida que cresce o tamanho do subcubo. Assim, um subcubo contido num subcubo maior não deve ser escolhido.

Note que os don't care states podem e devem ser usados para facilitar a obtenção de expressões mínimas, assumindo convenientemente os valores 0 ou 1. Neste exemplo, aos don't care states 14 e 15 é atribuído o valor 1, e o 3 assume valor 0.

Um implicante que corresponde a um subcubo não contido em nenhum subcubo maior é chamado de implicante-primo.

Um implicante-primo, além de implicar a função, não pode implicar nenhum outro produto de menor número de literais (subcubo maior) que também implique a função dada.

NOTAÇÃO UTILIZADA PARA IMPLICANTES

Qualquer implicante pode ser representado por um par de "rótulos" (números inteiros decimais), utilizados conjuntamente, como se segue :

REFERÊNCIA: é a célula base (de menor valor decimal) do subcubo.

REDUNDÂNCIA: é a soma dos pesos binários das variáveis que não aparecem no produto de literais (-).

Para exemplificar o uso desta notação e verificar a sua correspondência biunívoca com os termos da função, é apresentado na tabela II.4 o conjunto dos implicantes-primos da função F obtido em (II.25) e sua representação na notação introduzida.

Observe que esta notação é extremamente concisa e que, independente do número de variáveis da função, é sempre um par de números inteiros. Esta propriedade é fundamental na implementação computacional dos algoritmos.

Implicantes	Células	Binários w x y z	Referência	Redundância
w.x	12,13,14,15	1 1 - -	12	3
x.y'	4,5,12,13	- 1 0 -	4	9
w.y'.z'	8,12	1 - 0 0	8	4

Tabela II.4.- Equivalência de notações.

II.4.- DETECÇÃO DE FALHAS

Circuitos lógicos estão sujeitos a falhas em seu funcionamento, podendo ser, por exemplo, resultado da deterioração física de algum componente ou algum defeito no processo de fabricação.

Define-se uma falha em um circuito lógico como sendo qualquer alteração que modifique a característica lógica deste circuito. Um teste para uma determinada falha é um padrão de sinais que aplicados nos pontos controláveis (entradas externas ou entradas primárias) do circuito produz nos pontos observáveis (saídas externas ou saídas primárias) um valor lógico de acordo com a presença ou ausência de falha.

Diz-se que uma falha é detectável quando existir um vetor de entradas primárias que causa um erro em pelo menos uma de suas saídas primárias. Entendendo-se por erro um valor lógico distinto daquele do circuito sem defeitos.

As falhas podem ser modeladas de várias formas sendo as principais conhecidas por: falha fixa em Um, falha fixo em Zero, falha de curto-circuito entre conexões, etc.

O teste de circuitos digitais é realizado aplicando-se a este circuito sequências de estímulos nos terminais de entrada e observando-se as respostas nos terminais de saída. Comparando-se a sequência observada com a correspondente tabela verdade ou com a sequência produzida por um circuito sem falhas pode-se determinar a existência de erros.

Esses estímulos são denominados vetores de teste ou padrões de teste, onde cada estímulo é uma n-upla binária.

Cada padrão de teste tem associado a si um conjunto de falhas detectáveis. Diz-se neste caso que o padrão de teste "cobre" estas falhas.

Desta forma a um determinado conjunto de testes está associado uma correspondente cobertura de falhas do circuito.

Define-se como teste exaustivo a geração de todas as possíveis n-uplas da entrada. Este tipo de teste é geralmente muito longo, tornando-se impraticável para circuitos complexos.

As técnicas de geracão de padrões de teste têm como objetivo alcançar um alto grau de cobertura (próximo de 100%) com um número reduzido de padrões de testes distintos.

II.4.1.- Algoritmos de Geração de Teste

As técnicas de geração de testes mais usuais utilizam a descrição topológica do circuito, ou seja, a descrição dos seus elementos e suas interconexões para determinar os padrões de testes.

Os algoritmos para a geração de testes em circuitos combinacionais se baseiam no conceito de sensibilização de caminho (um caminho é qualquer ligação entre entradas e saídas do circuito, envolvendo as portas lógicas ao longo deste percurso), em que para se detectar uma falha é necessário ativá-la através de algum vetor de teste e propagá-la até a saída do circuito.

Diz-se que uma falha está ativada quando o padrão de teste causa no ponto com falha um valor lógico contrário ao imposto pela falha.

Uma falha se propaga até a saída se o padrão de teste implicar em alguma saída primária um valor lógico dependente da presença ou não da falha.

Vários algoritmos foram propostos para gerar padrões de testes em circuitos lógicos.

Em 1966 Roth [1,3,10] apresentou um algoritmo denominado algoritmo d.

Em 1981 Goel [2] propôs o PODEM (Path-oriented-decision making).

Em 1983 Fujiwara [4] apresentou o FAN (Fan-out oriented test generation algoritm).

Em 1988 Côrtes [5] apresentou o I-ALG (Um Algoritmo Incremental para a Geração de Padrão de teste).

Em 1989 Mendonça [11] apresentou o H-ALG (Um Gerador Hierárquico de Padrão de Teste).

O Algoritmo PODEM [2] é significativamente mais rápido que o algoritmo D e trata as portas lógicas ou-exclusivo com mais eficiência que este último.

Para gerar um teste o algoritmo utiliza uma enumeração implícita, semelhante às utilizadas para resolver problemas de programação inteira, em que todos os possíveis padrões de entrada externa são eventualmente examinados como candidatos a teste para uma dada falha.

Valores lógicos são atribuídos para as entradas primárias e a implicação para frente é executada.

Isto é, em cada passo do algoritmo é determinado o valor lógico do maior número de vértices do cubo teste que podem ser implicados de maneira única a partir da atribuição efetuada.

O Algoritmo de geração de teste FAN (Fan-out-oriented test generation algoritm) [4] é similar ao PODEM. Baseia-se, portanto, num processo de enumeração implícita.

Os algoritmos I-ALG e H-ALG baseiam-se na idéia de resolver pequenas células e depois combiná-las de uma forma incremental. As células podem ser portas lógicas simples, porta lógicas complexas etc.

Todos estes métodos são evoluções do algoritmo D e guardam com ele uma relação metodológica estreita.

Apresenta-se a seguir uma descrição sucinta do algoritmo D para ilustrar esta classe de algoritmos.

Algoritmo D [3]

Este algoritmo utiliza o conceito de sensibilização de caminhos múltiplos e a representação cúbica para descrever o circuito e suas falhas.

Para cada falha simples no circuito o algoritmo D determina se existe ou não um teste, isto é, determina se a falha é detectável ou não.

O símbolo D representa um sinal associado a um ponto do circuito lógico que assume o valor "1" no circuito normal e o valor "0" no circuito falho; $\bar{D}$ representa um sinal que assume o valor "0" no circuito normal e o valor "1" no circuito falho.

Define-se como Cubo primitivo de uma função lógica a qualquer implicante primo da função ou da função negada.

Cada cubo define a relação entre as entradas e a saída, representando a tabela verdade da porta lógica de maneira compacta.

A tabela II.5 apresenta os cubos primitivos de uma porta AND (figura II.6) tendo como entradas os vértices 1 e 2 e como saída o vértice 3.

1	2	3
0	X	0
X	0	0
1	1	1

Tabela II.5.- Tabela verdade da função AND na forma de cubos primitivos.

Figura II.6.- Porta lógica AND tendo como entrada os vértices 1 e 2 e como saída o vértice 3.

Na tabela II.5 o primeiro cubo indica que, se a entrada 1 assumir o valor "0", a saída 3 assumirá o valor "0", independentemente do valor da entrada 2.

A tabela II.6 apresenta os cubos primitivos de uma porta lógica equivalente ao circuito da figura II.7.

1	2	3	4
0	0	0	0
0	X	1	1
0	1	X	1
1	0	X	0
1	X	0	0
1	1	1	1

Tabela II.6.- Cubos primitivos da porta lógica equivalente ao circuito da figura II.7.

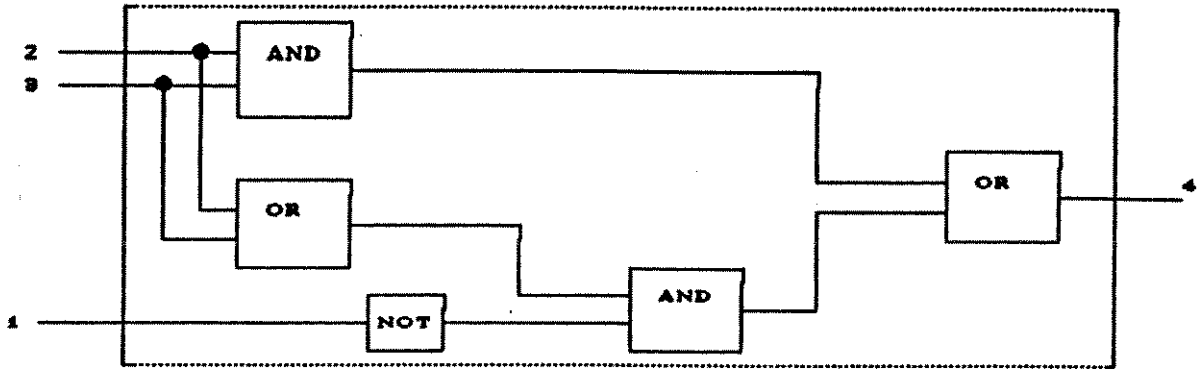


Figura II.7.- Circuito lógico combinacional de 3 entradas.

Com o intuito de analisar a propagação de uma falha ao longo de um dado circuito define-se a operação de intersecção entre os valores lógicos 0, 1 e X conforme a tabela II.7.

∩	0	1	X
0	0	D	0
1	$\bar{D}$	1	1
X	0	1	X

Tabela II.7.- Definição da operação de intersecção entre os valores lógicos 0,1,X.

É importante observar que a operação de intersecção não é comutativa, pois $0 \cap 1 = D$ e $1 \cap 0 = \bar{D}$.

Define-se cubos de propagação como os cubos que especificam as condições das demais entradas de uma porta lógica, necessárias para propagar-se falhas de uma ou mais entradas para a saída.

Os cubos de propagação de uma dada porta lógica podem ser obtidos pela intersecção de seus cubos primitivos que apresentam saídas distintas [1,3,10].

O cubo resultante da intersecção entre dois cubos primitivos é formado pela intersecção dos vértices correspondentes.

A tabela II.8 apresenta os cubos de propagação para a porta lógica AND onde o primeiro cubo indica que uma falha no vértice 1 propaga-se para a saída desde que o vértice 2 seja mantido em 1.

1	2	3
D	1	D
1	$\overline{D}$	D
D	D	D

Tabela II.8.- Cubos de propagação para a porta AND de duas entradas apresentada na figura II.6.

A tabela II.9 apresenta os cubos de propagação da porta lógica da figura II.7.

Uma falha simples ao se propagar define um caminho que recebe o nome de caminho de propagação ou caminho sensibilizado. Os valores lógicos ao longo deste caminho recebe o nome de erros.

No caso de caminhos divergentes (fan-out maior que 1) pode ocorrer erros múltiplos. Note que a propagação de uma falha simples através de caminhos múltiplos, necessita de cubos múltiplos de propagação para a sua análise.

1	2	3	4
0	0	D	D
0	D	0	D
$\overline{D}$	0	1	D
1	D	1	D
$\overline{D}$	1	0	D
1	1	D	D

A) Erros simples

1	2	3	4
D	0	D	D
$\overline{D}$	D	0	D
1	D	D	D
0	D	D	D
$\overline{D}$	D	1	D
$\overline{D}$	1	D	D

B) Erros duplos

1	2	3	4
D	D	D	D
$\overline{D}$	D	D	D
$\overline{D}$	D	$\overline{D}$	D
$\overline{D}$	$\overline{D}$	D	D

C) Erros triplos

Tabela II.9.- Cubos de propagação da porta lógica apresentada na figura II.7.

Define-se cubos de falha (de saída) como qualquer cubo primitivo utilizado para excitar o vértice com falha. Nestes cubos, são especificadas as condições das entradas para produzir um sinal D (saída fixa em 0) ou $\overline{D}$ (saída fixa em 1).

A tabela II.10 apresenta os cubos de falha para a porta AND da figura II.6.

1	2	3
0	X	$\bar{D}$
X	0	$\bar{D}$
1	1	D

Tabela II.10.- Cubos de falha para a porta AND apresentada na figura II.6.

O algoritmo D trata de circuitos combinacionais com falhas simples.

Para cada porta lógica do circuito é possível a análise de falhas "fixo em zero" e "fixo em um" de sua saída.

Para análise de falhas em uma dada entrada de uma porta lógica pode-se definir portas lógicas fictícias nestas entradas.

A cada saída é associado um vértice em um cubo, descrevendo o estado do circuito, denominado cubo teste. Portanto pode-se associar a cada circuito uma n-upla descrevendo o estado lógico da saída de cada uma de suas portas lógicas.

A definição de um teste para uma dada falha consiste, no algoritmo D, na determinação de um cubo teste que contenha :

1. D (falha fixa em 0) ou $\bar{D}$ (falha fixa em 1) na posição da falha;
2. Em pelo menos uma saída externa os valores D ou $\bar{D}$;
3. Nas demais posições deve-se encontrar os valores lógicos $D, \bar{D}, 1, 0$ ou X.

Estes valores devem ser consistentes com os cubos primitivos e de propagação de cada porta lógica do circuito.

Com o intuito de obtenção de um cubo teste consistente define-se a intersecção D de um cubo teste com um cubo primitivo ou com um cubo de propagação conforme a tabela II.11 . Note que a operação intersecção D é comutativa e associativa, quando definida.

Ω	0	1	X	D	$\overline{D}$
0	0	∅	0	ϕ	ϕ
1	∅	1	1	ϕ	ϕ
X	0	1	X	D	$\overline{D}$
D	ϕ	ϕ	D	μ	λ
$\overline{D}$	ϕ	ϕ	$\overline{D}$	λ	μ

Tabela II.11.- Definição da operação intersecção D entre cubos.

O simbolo ϕ indica a impossibilidade de intersecção, isto é , produziria cubos teste inconsistentes.

Se na intersecção de um cubo teste com um cubo de propagação resultar em cubo teste com símbolos 0,1,X,D,$\overline{D}$ e μ, então define-se a posteriori, D Ω D = D e $\overline{D}$ Ω $\overline{D}$= $\overline{D}$.

Se resultar nos símbolos 0,1,X,D, $\overline{D}$ e λ então troca-se no cubo de propagação D por $\overline{D}$ e vice-versa e repete-se a operação de intersecção.

Se resultar nos símbolos 0,1,X,D,$\overline{D}$,λ e μ então a intersecção é indefinida e o cubo teste é inconsistente.

Descrição do algoritmo

1. Inicializa-se o cubo teste com todos os vértices no valor X, salvo o vértice com falha, no valor D (falha fixo em 0) ou $\overline{D}$ (falha fixo em 1).

2. Escolhe-se um dos cubos de falha, para o vértice falho do circuito e realiza-se a intersecção D com o cubo teste definido no passo 1.

Esta escolha inicial é arbitrária; se resultar durante a execução do algoritmo em um cubo teste inconsistente deve-se retornar e considerar outra possível escolha. Este processo é denominado de retorno.

O retorno deve ser efetuado até que todas as possíveis escolhas tenham sido consideradas. Caso não haja mais escolha a falha é não detectável.

3. Inicia-se o processo da determinação dos caminhos múltiplos sensibilizados (caminhada para frente) que ligam a falha a uma ou mais saídas externas (pontos observáveis).

Para isto determina-se um vértice ativo, isto é , escolhe-se um dentre todos os novos vértices com valor D ou $\overline{D}$.

A cada vértice ativo, está associado um conjunto de portas lógicas

subsequentes (lista de fan-out).

A propagação deve ser realizada através de todas as portas lógicas (caminhos múltiplos).

Para isto seleciona-se para cada porta lógica subsequente, um de seus cubos de propagação e realiza-se a operação de intersecção D, resultando em um cubo teste que deve ser consistente.

Se o cubo teste resultante não for consistente deve-se realizar uma nova escolha de cubo de propagação. Se isto ocorrer para todos os cubos de propagação, então a falha não pode se propagar através da referida porta lógica.

As portas lógicas são numeradas por níveis, da entrada para a saída.

Uma vez esgotadas todas as portas lógicas associadas ao fan-out do vértice ativo, avança-se para tratar um novo vértice ativo do mesmo nível.

Uma vez esgotados todos os vértices ativos de um dado nível, avança-se para o próximo nível.

4. O passo 3 é repetido até que os símbolos D ou $\bar{D}$ alcancem as saídas externas do circuito.

5. O último passo do algoritmo recebe o nome de operação de consistência (caminhada para trás); e deve resultar na definição de um cubo teste em que todos os vértices com valores 0 e 1 devem ser testados na sua consistência.

Para isto, iniciando-se do nível mais alto para o mais baixo, trata-se todas as portas lógicas antecessoras com valores 0 ou 1 em suas saídas. Isto é feito através da intersecção do cubo teste com o cubo primitivo de cada uma das portas lógicas antecessoras.

Uma vez esgotada uma porta lógica (lista de fan-in), trata-se as demais portas de mesmo nível.

Uma vez esgotado este nível, trata-se o nível anterior.

A ocorrência de cubos testes inconsistentes exige a operação retorno que equivale a testar um outro cubo primitivo para a porta em questão.

Ao atingir as entradas externas com um cubo teste consistente ter-se-á obtido um teste para a falha.

Se isto não for possível a falha é não testável.

Com o propósito de tornar mais claro os passos do algoritmo d, serão apresentados a seguir alguns exemplos.

II.4.2.- EXEMPLOS

Exemplo 01. Para o circuito da figura II.8 considere a falha f-e-1 no vértice 1.2.

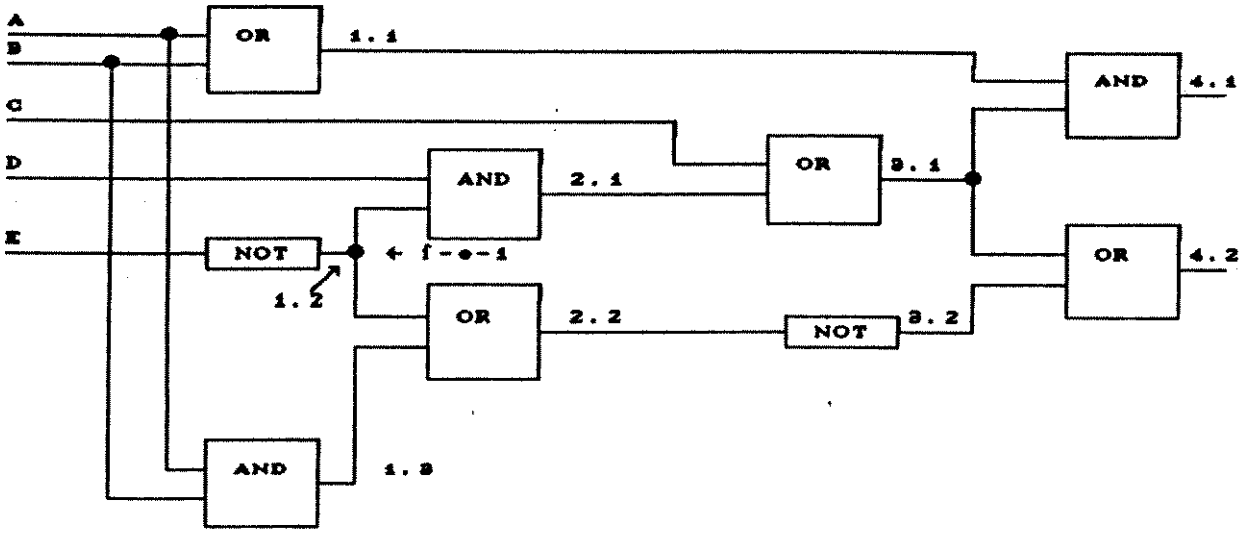


Figura II.8.- Circuito combinacional contendo portas OR, AND e NOT

As tabelas II.12, II.13 e II.14 apresentam os cubos primitivos e os cubos de propagação para as portas OR, AND e NOT respectivamente.

E1	E2	S
1	X	1
X	1	1
0	0	0
D	0	D
0	D	D
D	D	D

TABELA II.12.- Cubos primitivos e cubos de propagação para a porta OR.

E1	E2	S
0	X	0
X	0	0
1	1	1
D	1	D
1	D	D
D	D	D

TABELA II.13.- Cubos primitivos e cubos de propagação para a porta AND.

E	S
1	0
0	1
D	$\bar{D}$
$\bar{D}$	D

TABELA II.14.- Cubos primitivos e cubos de propagação para a porta NOT.

A tabela II.15 apresenta os cubos teste para cada passo do algoritmo.

LINHA	DESCRIÇÃO	A	B	C	D	E	1.1	1.2	1.3	2.1	2.2	3.1	3.2	4.1	4.2
1	FALHA							$\bar{D}$							
2	CT-0					1		$\bar{D}$							
3	CP-2.1				1			$\bar{D}$		$\bar{D}$					
4	CT-1				1	1		$\bar{D}$		$\bar{D}$					
5	CP-2.2							$\bar{D}$	0		$\bar{D}$				
6	CT-2				1	1		$\bar{D}$	0	$\bar{D}$	$\bar{D}$				
7	CP-3.1			0						$\bar{D}$		$\bar{D}$			
8	CT-3			0	1	1		$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$			
9	CP-3.2										$\bar{D}$		D		
10	CT-4			0	1	1		$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$	D		
11	CP-4.1						1					$\bar{D}$		$\bar{D}$	
12	CT-5			0	1	1	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$	D	$\bar{D}$	
13	CP-4.2											$\bar{D}$	0		$\bar{D}$
14	CT-6			0	1	1	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$	ϕ	$\bar{D}$	$\bar{D}$
15	CT-5			0	1	1	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$	D	$\bar{D}$	
16	1.1	1					1								
17	CT-6	1		0	1	1	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$		$\bar{D}$	
18	1.3		0						0						
19	CT-7	1	0	0	1	1	1	$\bar{D}$	0	$\bar{D}$	$\bar{D}$	$\bar{D}$		$\bar{D}$	

TABELA II.15.- Cubos teste para cada passo do algoritmo.

A seguir serão descritas todas as linhas da tabela II.15.

Na linha 1 inicializa-se o cubo teste, fixando o vértice 1.2 com o valor lógico $\bar{D}$ (falha fixa em 1) e os outros vértices do circuito com o valor lógico X. Para tornar o cubo teste mais legível, o valor lógico X é representado por

espaços em branco.

Na linha 2 é escolhido um cubo de falha para a porta 1.2. Esta linha da tabela recebe o rótulo de cubo teste 0 (CT-0). O passo seguinte consiste em inserir na lista de fanout todas as portas subsequentes ao vértice ativo do CT-0 e efetuar a propagação (caminhada para frente) do sinal falho através destas portas. O vértice ativo para o CT-0 possue como portas subsequentes o conjunto (2.1 , 2.2).

Na linha 3 é selecionado um dos cubos de propagação para a porta 2.1.

Na linha 4 é efetuada a operação intersecção D entre o cubo teste CT-0 e o cubo de propagação da porta 2.1, resultando no cubo teste CT-1. O resultado intermediário dos cubos teste, onde $\bar{D} \Omega \bar{D} = \mu$, deve ser suprimido.

Na linha 5 é selecionado um dos cubos de propagação para a porta 2.2.

Na linha 6 é efetuada a operação intersecção D entre o cubo teste CT-1 e o cubo de propagação da porta 2.2, resultando no cubo teste CT-2.

Após efetuada a propagação do sinal por todas as portas da lista de fanout de um mesmo nível deve-se propagar o sinal falho pelas portas subsequentes ao vértice ativo do último cubo teste.

Os vértices ativos para o nível seguintes são as portas 2.1 e 2.2 e as portas subsequentes a estes vértices formam a lista de fanout representada pelo conjunto (3.1 , 3.2).

A linha 7 apresenta um dos cubos de propagação para a porta 3.1.

A linha 8 apresenta o cubo teste CT-3 formado pela intersecção do cubo teste CT-2 com o cubo de propagação da porta 3.1.

A linha 9 apresenta o cubo de propagação da porta 3.2.

A linha 10 apresenta o cubo teste CT-4 resultado da operação intersecção D do cubo teste anterior com o cubo de propagação da porta 3.2.

O cubo teste CT-4 possui como vértices ativo as portas 3.1 e 3.2. A lista de fanout é formada pelos conjunto (4.1 e 4.2).

A linha 11 apresenta o cubo de propagação para a porta 4.1 e a linha 12 o resultado da intersecção deste cubo com o cubo teste anterior, resultando no cubo teste CT-5.

A linha 13 apresenta o cubo de propagação para a porta 4.2 e a linha 14 o resultado da intersecção deste cubo com o cubo teste anterior. Note que foi gerada uma inconsistência nos valores lógicos atribuidos ao vértice 3.2 representada pelo símbolo ϕ. Neste caso deve-se retornar ao último cubo teste consistente e selecionar outro cubo de propagação para a porta 4.2. Na tabela II.15 um cubo teste inconsistente é indicado por um contorno tracejado.

Pela tabela II.13 pode-se verificar que a porta lógica 4.2 não possue

nenhum cubo de propagação capaz de propagar o sinal falho para a sua saída. Como o sinal falho atingiu uma das saídas do circuito, deve-se descartar a propagação por este caminho.

O cubo teste CT-5 não possue nenhum vértice ativo, visto que o sinal D atingiu uma das saídas do circuito.

Inicia-se a operação de consistência (caminha para trás), efetuando-se a operação intersecção D entre o cubo teste CT-5 com os cubos primitivos das portas antecessoras a porta 4.1 com valores 0 ou 1 em suas saídas. A operação é efetuada do nível mais alto para o nível mais baixo. Na tabela II.15 o início da caminhada para trás é indicada por um traço duplo.

A linha 16 apresenta o cubo primitivo da porta 1.1 que produz o sinal 1 em sua saída.

A linha 17 apresenta o cubo teste CT-6; resultado da operação intersecção D entre o cubo primitivo da porta 1.1 e o cubo teste anterior.

A linha 18 apresenta o cubo primitivo da porta 1.3 e a linha 19 o cubo teste CT-7.

Do cubo teste CT-7 pode-se observar que foram atingidas as entradas externas com um cubo teste consistente.

O cubo teste CT-7 define o teste (1,0,1,1,1), que detecta a falha f-e-1 no vértice 1.2 do circuito.

Note que este teste também detecta as falhas fixo em 1 nos vértices 2.1, 2.2, 3.1 e 4.1.

Exemplo 02. Para o circuito da figura II.9 considere a falha f-e-0 na porta 1.4.

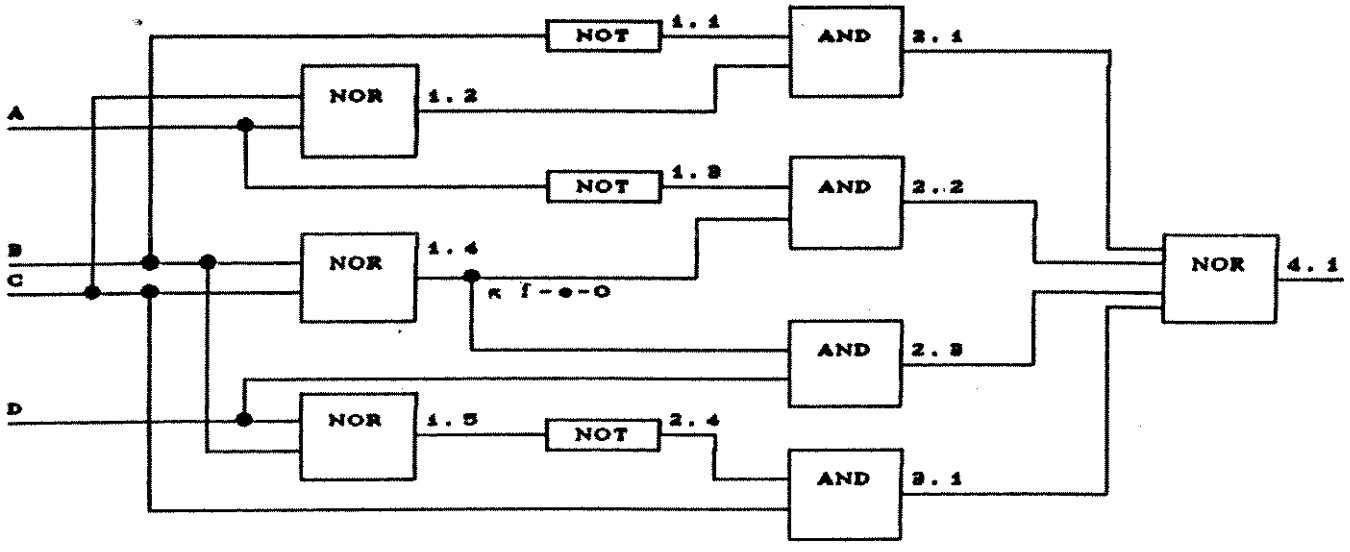


Figura II.9.- Circuito combinacional contendo portas AND, NOR e NOT.

As tabelas II.16 e II.17 apresentam os cubos primitivos e os cubos de propagação para as portas NOR com duas e quatro entradas repectivamente.

E1	E2	S
1	X	0
X	1	0
0	0	1
D	0	D
0	D	D
D	D	D

TABELA II.16.- Cubos primitivos e cubos de propagação para as portas NOR com duas entradas.

A tabela II.18 apresenta os cubos teste para cada passo do algoritmo.

Os passos para este exemplo são os mesmos do exemplo anterior, portanto não serão detalhados.

A linha 7 mostra que a propagação pela porta 4.1 utilizou-se de um cubo de propagação duplo.

E1	E2	E3	E4	S
1	X	X	X	0
X	1	X	X	0
X	X	1	X	0
X	X	X	1	0
0	0	0	0	1
D	0	0	0	D
0	D	0	0	D
0	0	D	0	D
0	0	0	D	D
D	D	0	0	D
D	0	D	0	D
D	0	0	D	D
0	D	D	0	D
0	D	0	D	D
0	0	D	D	D
D	D	D	0	D
D	D	0	D	D
0	D	D	D	D
D	D	D	D	D

TABELA II.17.- Cubos primitivos e cubos de propagação para a porta NOR de 4 entradas.

A linha 14 mostra a inconsistência nos valores atribuidos ao vértice C no processo de caminhada para trás. Isto provocou um retorno ao cubo teste CT-5 (último cubo teste consistente) e a escolha de um outro cubo primitivo para a porta 1.2 que produz em sua saída o valor lógico 0.

A linha 17 resultou em um cubo teste consistente em que a combinação dos valores lógicos dos vértices de entrada produz o teste (1,0,0,1) que detecta a falha f-e-0 no vértice 1.4. Note que este teste também detecta as falhas fixo em zero nos vértices 2.2 e 2.3 e falhas fixo em 1 no vértice 4.1.

LIN	DESCRI	A	B	C	D	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	3.1	4.1
1	FALHA								D							
2	CT-0		0	0					D							
3	CP-2.2							1	D			D				
4	CT-1		0	0				1	D			D				
5	CP-2.3				1				D				D			
6	CT-2		0	0	1			1	D			D	D			
7	CP-4.1										0	D	D		0	$\bar{D}$
8	CT-3		0	0	1			1	D		0	D	D		0	$\bar{D}$
9	2.1						0				0					
10	CT-4		0	0	1		0	1	D		0	D	D		0	$\bar{D}$
11	3.1			0											0	
12	CT-5		0	0	1		0	1	D		0	D	D		0	$\bar{D}$
13	1.2			1			0									
14	CT-6			ϕ	1		0	1	D		0	D	D		0	$\bar{D}$
15	CT-5		0	0	1		0	1	D		0	D	D		0	$\bar{D}$
16	1.2	1					0									
17	CT-6	1	0	0	1		0	1	D		0	D	D		0	$\bar{D}$

TABELA II.18.- Cubos teste para cada passo do algoritmo.

Exemplo 03. Para o circuito da figura II.10 considere a falha f-e-1 na entrada bf da porta 3.1.

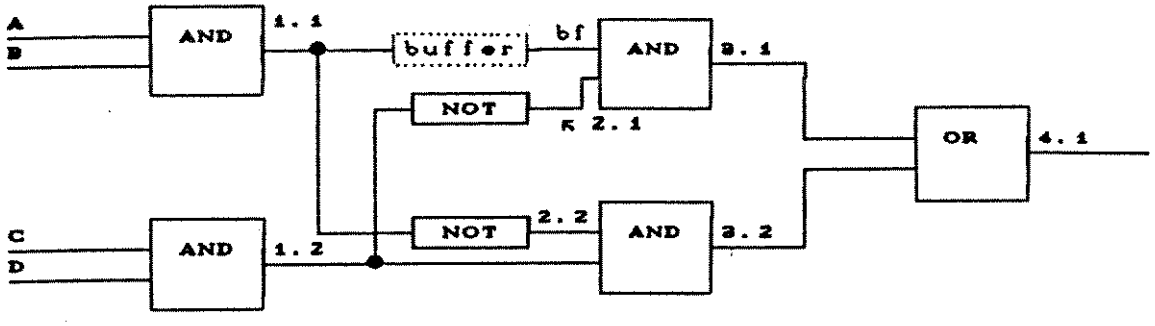


Figura II.10.- Circuito contendo portas OR, AND, NOT.

Para analisar falhas de entrada nas portas do circuito, deve-se considerar a existência de um "buffer" imaginário e gerar a falha na saída deste "buffer", que consequentemente é uma falha na entrada da porta subsequente a ele.

A tabela II.19 apresenta os cubos teste para cada passo do algoritmo.

LIN	DESCRI	A	B	C	D	1.1	1.2	bf	2.1	2.2	3.1	3.2	4.1
1	FALHA							$\bar{D}$					
2	CT-0					0		$\bar{D}$					
3	CP-3.1							$\bar{D}$	1		$\bar{D}$		
4	CT-1					0		$\bar{D}$	1		$\bar{D}$		
5	CP-4.1										$\bar{D}$	0	$\bar{D}$
6	CT-2					0		$\bar{D}$	1		$\bar{D}$	0	$\bar{D}$
7	3.2									0		0	
8	CT-3					0		$\bar{D}$	1	0	$\bar{D}$	0	$\bar{D}$
9	2.2					1				0			
10	CT-4					ϕ		$\bar{D}$	1	0	$\bar{D}$	0	$\bar{D}$
11	CT-3					0		$\bar{D}$	1	0	$\bar{D}$	0	$\bar{D}$
12	CT-2					0		$\bar{D}$	1		$\bar{D}$	0	$\bar{D}$
13	3.2						0					0	
14	CT-3					0	0	$\bar{D}$	1		$\bar{D}$	0	$\bar{D}$
15	2.2					0				1			
16	CT-4					0	0	$\bar{D}$	1	1	$\bar{D}$	0	$\bar{D}$
17	2.1						0		1				
18	CT-5					0	0	$\bar{D}$	1	1	$\bar{D}$	0	$\bar{D}$
19	1.2			0			0						
20	CT-6			0		0	0	$\bar{D}$	1	1	$\bar{D}$	0	$\bar{D}$
21	1.1		0			0							
22	CT-7		0	0		0	0	$\bar{D}$	1	1	$\bar{D}$	0	$\bar{D}$

Tabela II.19.- Cubos teste para cada passo do algoritmo.

Neste exemplo mostrou-se a necessidade de se considerar um buffer imaginário para se gerar falhas de entrada.

Pela linha 22 da tabela II.19 pode-se verificar que a falha fixa em 1 na entrada bf da porta 3.1 pode ser detectada pelos vetores de teste 0000, 1000, 1001 ou 0001.

II.5.- CONCLUSÃO

Neste capítulo foram apresentados os conceitos básicos da álgebra booleana e as notações utilizadas em todo o trabalho.

As funções booleanas foram definidas afim de descrever as propriedades dos circuitos combinacionais.

A apresentação da secção II.4 (detecção de falha) forneceu ao leitor a base necessária para o entendimento do método de minimização através da geração de padrão de teste tratado no capítulo III.

II.6.- REFERÊNCIAS BIBLIOGRÁFICAS

[1] H. Y. Chang, E. Manning, G. Metze
Fault Diagnosis of Digital Systems.
Robert E.K. Publishing Company - 1974

[2] P. Goel
An Implicit Enumeration Algorithm to Generate tests for combinational logic circuits.
IEEE Trans. on Computers, Vol. C-30, No.3,March 1981 - pp 215-222

[3] J. P. Roth
Diagnosis of Automata Failures: A calculus and A method.
IBM Journal - july 1966

[4] H. Fujiwara, T. Shimino.
On the acceleration of teste Generation Algorithms.
IEEE Trans. on Computers, vol.C-32, No.12, December 1983

[5] M. L. Côrtes, J. M. F. Mendonça, S. A. O. Andrade
I-ALG: Um algoritmo incremental para geração de padrão de teste.
III Congresso Brasileiro da Sbmicro, S.Paulo, Julho de 1988, pp 427-436

[6] Z. Kohavi
Switching and Finite Automata Theory
McGraw Hill - 1970

[7] F. Hill, & G. Peterson
Introduction to Switching Theory & Logical Design
John Wiley & Sons - 1981

[8] Y. Yano, D. Camilo, & J.B.T. Yabuti-uti
Circuitos lógicos: Teoria e laboratório: Engenharia Eletrônica
Ed. Livraria Ciência e Tecnologia - 1984

[9] M. C. G. Madureira
Contribuição à Análise e Síntese de Circuitos Digitais.
Tese de Mestrado - FEE - UNICAMP, Maio de 1987

[10] M. A. Breuer, A. D. Friedman
Diagnosis & Reliable Design of Digital Systems.
Computer Science Press, inc - 1976

[11] J. M. F. Mendonça, M. L. Côrtes, S. A. D. Andrade
H-ALG: "Um Gerador Hierárquico de Padrão de teste.
IV Congresso Brasileiro da SBmicro, Porto Alegre, Julho de 1989, pp 441-449

C A P Í T U L O III

MINIMIZAÇÃO DE FUNÇÕES BOOLEANAS

III.1.- INTRODUÇÃO

A disseminação do uso de PLA's (Estruturas Lógicas Programáveis) no projeto de circuitos digitais mantém e amplia a relevância da pesquisa em algoritmos de minimização de funções booleanas. Neste capítulo são descritos três métodos de minimização de funções booleanas: "Quine-McCluskey" [1], "Caruso" [2],[5] e "Cobertura Irredundante através da Geração de Padrão de Teste" (Geração) [3].

O algoritmo de "Quine-McCluskey" foi implementado para servir de referência e para validar os outros dois métodos.

O método de "Caruso" foi escolhido por ter se mostrado mais eficiente que o de "Quine-McCluskey" e por ser um método bastante recente na literatura.

O método "Geração" é baseado no algoritmo Podem [4], inicialmente proposto para detecção de falhas.

O problema de simplificar funções booleanas consiste em obter a partir da tabela verdade uma expressão booleana mínima de acordo com algum critério de custo.

Grande parte dos métodos existentes possui duas fases: a obtenção dos implicantes primos, e a cobertura irredundante, onde a partir do conjunto de implicantes primos são obtidos os implicantes essenciais para a realização da função.

O algoritmo de "Quine-McCluskey" também conhecido como método tabular, é o mais clássico desta classe de algoritmos. O algoritmo de "Caruso" utiliza-se de grafos com a estrutura de árvores e através de uma seleção local de implicantes, procura sanar o principal problema do algoritmo de "Quine-McCluskey", que é a geração de todos os implicantes para a obtenção dos

implicantes primos essenciais para a realização da função. O algoritmo "Geração" determina uma cobertura irredundante de uma função booleana a partir de uma adaptação do algoritmo Podem, utilizando a técnica de enumeração implícita.

A idéia básica do algoritmo "Geração" é partir de uma cobertura inicial da função e através de falhas f-e-1 (fixa em 1) nos AND's e falhas f-e-0 (fixa em 0) no OR, detectar e eliminar as possíveis redundâncias da cobertura inicial.

O método tem como principais características o uso restrito de memória e simplicidade de implementação.

No final do capítulo os três algoritmos são comparados através de vários exemplos.

A simplificação de funções booleanas é parte essencial na redução dos custos da realização de circuitos lógicos e aritméticos binários e é por este motivo que este trabalho aprofunda esta questão. Entretanto, é importante ressalvar a importância da análise da lógica de multi-valores que deverá ter grande impacto na realização das funções lógicas [8].

III.2.- ALGORITMO DE QUINE-McCLUSKEY

O método de Quine-McCluskey simplifica funções booleanas no sentido de minimizar os custos dos circuitos digitais que as sintetizam.

Como expressão mínima da função é adotada a forma de soma de produtos, conforme o critério de custos definido no capítulo II.

A idéia fundamental do método é a aplicação repetida da propriedade:

$$xy + xy' = x (y + y') = x$$

Exemplo 1. Para minimizar a função

F1(w,x,y,z) = S(0,1,8,9), mostrada na figura III.1, tem-se:

F1 = w'x'y'z' + w'x'y'z + wx'y'z' + wx'y'z

F1 = w'x'y' (z + z') + wx'y' (z + z') = w'x'y' + wx'y'

F1 = x'y' (w + w') = x'y'

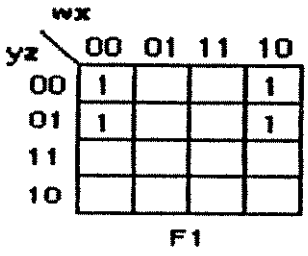


Figura III.1.- Mapa de Karnaugh para F1 = S(0,1,8,9)

III.2.1.- SISTEMATIZAÇÃO DO MÉTODO

O método de Quine-McCluskey consiste de duas fases: a geração de implicantes primos e a cobertura dos mintermos.

Com os passos definidos a seguir, a primeira fase do método se torna sistemática e mais facilmente adaptada aos procedimentos computacionais.

PRIMEIRA FASE: GERAÇÃO DE IMPLICANTES PRIMOS

1.-Separe os mintermos e don't care states em "caixas", sendo que na mesma caixa ficam os termos com o mesmo número de dígitos '1' na sua forma binária.Este número de 1's é o índice da caixa.Esta operação define o GRUPO 0.

2.- Para todo i no intervalo [0..N] (onde N é o número de variáveis da função), compare cada termo da caixa de índice i com todos os termos da caixa i+1. Combine-os de acordo com o teorema $xy+xy' = x$, isto é, dois termos são combináveis se pertencerem a caixas adjacentes e diferirem por apenas um dígito na representação binária. Na representação decimal da referência dois termos são combináveis se diferirem por uma potência de 2 (1,2,4,8,...) e tiverem a mesma redundância. Além disso, o termo da caixa de índice superior (i+1) deve ser sempre maior que o termo da caixa de índice inferior (i). Devem ser marcados todos os termos combinados pelo menos uma vez. Esta operação gera um novo GRUPO.

3.- Combine da mesma forma os termos gerados no passo anterior .

4.- O processo acima (3) continua até que não haja mais combinação possível. Os termos não marcados constituem o conjunto de implicantes-primos da função dada.

Exemplo 2. Aplicando-se a primeira fase do método a função

F2(w,x,y,z) = S(0,1,2,5,7,8,9,10,13,15) (III.1)

obtém-se a TABELA III.1 onde pode-se observar que :

.- o grupo é dado pelo número de combinações ocorridas, ou seja, pelo

número de posições irrelevantes ("-") na representação binária;

.- as caixas são dadas pelo número de 1's na forma binária;

.- o grupo 0 é formado sempre pelos próprios mintermos e don't care states da função .

.- na geração dos demais grupos, pode-se produzir um mesmo implicante mais de uma vez, porém ele constará apenas uma vez da tabela.

CX.	DECIMAIS	BINARIOS	TICK
0	0,0	0 0 0 0	T
1	1,0	0 0 0 1	T
1	2,0	0 0 1 0	T
1	8,0	1 0 0 0	T
2	5,0	0 1 0 1	T
2	9,0	1 0 0 1	T
2	10,0	1 0 1 0	T
3	7,0	0 1 1 1	T
3	13,0	1 1 0 1	T
4	15,0	1 1 1 1	T

GRUPO 0

CX.	DECIMAIS	BINARIOS	TICK
0	0,1	0 0 0 -	T
0	0,2	0 0 - 0	T
0	0,8	- 0 0 0	T
1	1,4	0 - 0 1	T
1	1,8	- 0 0 1	T
1	2,8	- 0 1 0	T
1	8,1	1 0 0 -	T
1	8,2	1 0 - 0	T
2	5,2	0 1 - 1	T
2	5,8	- 1 0 1	T
2	9,4	1 - 0 1	T
3	7,8	- 1 1 1	T
3	13,2	1 1 - 1	T

GRUPO1

CX.	DECIMAIS	BINARIOS	TICK
0	0,9	- 0 0 -	F
0	0,10	- 0 - 0	F
1	1,12	- - 0 1	F
2	5,10	- 1 - 1	F

GRUPO2

Tabela III.1.- Tabela para minimização de F2(w,x,y,z) =
= S(0,1,2,5,7,8,9,10,13,15)

Destaca-se neste exemplo que apenas os termos do grupo 2 não foram combinados e portanto não foram marcados (TICK = FALSE) , logo são eles que formam o conjunto dos implicantes-primos da função F2 :

w x y z

A = (0,9) = - 0 0 - = x'y'

B = (0,10) = - 0 - 0 = x'z'

C = (1,12) = - - 0 1 = y'z

D = (5,10) = - 1 - 1 = xz

F2(w,x,y,z) = x'y' + x'z' + y'z + xz (III.2)

O custo inicial da função, dado pela soma de todos os mintermos é 50; note a redução do custo para 12, apenas pela obtenção dos implicantes primos.

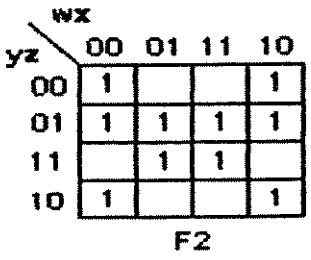


Figura III.2.- Mapa de Karnaugh de F2 = S(0,1,2,5,7,8,9,10,13,15)

III.2.2.- MAPA DOS IMPLICANTES-PRIMOS

De acordo com o método, após a obtenção do conjunto de todos os implicantes-primos da função, monta-se uma tabela na qual as linhas são os implicantes-primos e as colunas são os mintermos da função. Os estados irrelevantes (don't care states) da função não aparecem nesta tabela, uma vez que não precisam ser cobertos pela expressão mínima da função. Esta tabela é chamada de Mapa dos implicantes-primos da função.

A tabela é montada colocando-se uma marca (X) nas intersecções entre os implicantes-primos e os mintermos, indicando quais mintermos são cobertos por cada um dos implicantes-primos.

O problema resume-se então em selecionar um subconjunto mínimo de linhas da tabela (implicantes-primos) de modo que cada coluna tenha ao menos uma marca neste subconjunto, ou seja, todos os mintermos são cobertos por este subconjunto de implicantes-primos. Tal subconjunto deve ser o que possui número mínimo de literais em relação a todos os demais que possam ser selecionados (critério de custo 3 - capítulo II).

Para exemplificar, é montado a seguir o mapa dos implicantes-primos de F2, com base no seu conjunto de implicantes-primos obtidos na equação III.2. Este mapa é mostrado na figura III.3 Note no mapa que os mintermos 2, 7, 10 e 15 contêm uma única marca. Os implicantes-primos que cobrem estes mintermos são denominados IMPLICANTES PRIMOS ESSENCIAIS e aparecem obrigatoriamente na fórmula mínima da função.

IMPLICANTES	mintermos de F2									
	0	1	2	5	7	8	9	10	13	15
(0, 9)	X	X				X	X			
(0,10)	X		X			X		X		
(1,12)		X		X			X		X	
(5,10)				X	X				X	X

Figura III.3.- Mapa dos implicantes-primos de F2 .

Este mapa será reduzido com a retirada de linhas por serem consideradas ESSENCIAIS e de colunas (mintermos) por terem sido cobertas, dando origem à fórmula mínima da função.

No algoritmo são definidos diferentes NIVEIS de essenciais, a saber:

NIVEL 0 : São os essenciais obtidos logo na montagem do mapa, ou seja, pela regra básica, que é cobrir um mintermo que não seja coberto por nenhum outro implicante-primo.

NIVEL n : São os implicantes-primos considerados essenciais pela mesma regra, porém após n reduções do mapa.

Neste exemplo, os essenciais de nível 0 são os implicantes primos (0,10) e (5,10), e a tabela da figura III.3 é reduzida eliminando-se as linhas correspondentes a estes implicantes-primos e as colunas referentes a todos os mintermos cobertos por eles. Assim é obtida a tabela da figura III.4:

IMPLICANTES	1	9
(0, 9)	X	X
(1,12)	X	X

Figura III.4.- Mapa reduzido dos implicantes-primos de F2.

Da figura III.4, observa-se que falta cobrir apenas os mintermos 1 e 9, o que pode ser feito através dos implicantes-primos (0,9) ou (1,12). Neste caso qualquer das escolhas resulta numa expressão mínima para a função F2, já que ambos os implicantes possuem o mesmo número de literais (pois pertencem ao mesmo grupo), e o implicante escolhido é dito essencial nível 1 .

Logo, para esta função são obtidas 2 expressões mínimas:

$$F2 = (0,10) + (5,10) + (0,9) \qquad \text{CUSTO} = 9$$
$$= x'z' + xz + x'y' \qquad \text{(III.3)}$$

ou

$$F2 = (0,10) + (5,10) + (1,12) \qquad \text{CUSTO} = 9$$

$= x'z' + xz + y'z$

Observe, a redução dos custos, inicialmente 50, com a obtenção dos implicantes primos passa a 12, e com a redução do mapa cai para 9.

Como foi visto na seção II.3.1, algumas vezes a forma de produto de somas pode ser mais vantajosa e a função deve ser minimizada a partir de seus maxtermos. Entretanto, também foi mostrado que o custo obtido a partir dos maxtermos é equivalente ao custo obtido a partir dos mintermos da função negada.

Não há como se saber a priori se é ou não vantajosa a utilização de produto de somas ou soma de produtos, assim, deve-se fazer a minimização das duas funções, F e F' (F negada). Para F2 do exemplo 1, tem-se :

$F2' = S(3,4,6,11,12,14)$

A obtenção dos implicantes primos de F2' é mostrada na figura III.5:

GRUPO 0

CX.	DECIMAIS	BINARIOS	TICK
1	4,0	0 1 0 0	T
2	3,0	0 0 1 1	T
2	6,0	0 1 1 0	T
2	12,0	1 1 0 0	T
3	11,0	1 0 1 1	T
3	14,0	1 1 1 0	T

GRUPO 1

CX.	DECIMAIS	BINARIOS	TICK
1	4,2	0 1 — 0	T
1	4,8	— 1 0 0	T
2	3,8	— 0 1 1	F
2	6,8	— 1 1 0	T
2	12,2	1 1 — 0	T

GRUPO2

CX.	DECIMAIS	BINARIOS	TICK
1	4,10	— 1 — 0	F

Figura III.5.- Minimização de F2'.

Da figura III.5 nota-se que só existem dois implicantes primos para F2', (3,8) e (4,10), e que ambos são essenciais.

Assim, F2'= (3,8) + (4,10) = $x'yz + xz'$ com custo = 7, portanto mais barato que F2. Logo, neste caso, é vantagem minimizar F2 usando produto de somas (maxtermos) ao invés de mintermos. O resultado é :

$$F2 = (x + y' + z').(x' + z)$$

III.3.- FUNÇÕES NÃO COMPLETAMENTE ESPECIFICADAS

Para funções incompletamente especificadas, ou seja, funções onde aparecem estados irrelevantes (don't care states), o algoritmo de Quine-

McCluskey se adapta perfeitamente:

- A fase de geração dos implicantes primos trata os don't care da mesma forma que trata os mintermos, combinando-os da mesma forma;

- Na fase de cobertura, os don't care são simplesmente ignorados, não fazendo parte do mapa de implicantes primos, pois não necessitam ser obrigatoriamente cobertos.

Exemplo 3. Aplicar o algoritmo para a função

$$F3(v,w,x,y,z) = S(13,15,17,18,19,20,21,23,25,27,29,31) + D(1,2,12,24) \quad (III.4)$$

O mapa de Karnaugh de F3 é mostrado na figura III.6. Seus implicantes primos são obtidos na figura III.7. Na figura III.8 monta-se o mapa dos implicantes primos de F3 e a figura III.9 é este mapa reduzido.

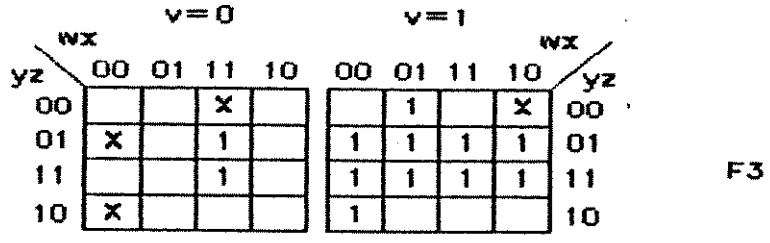


Figura III.6.- Mapa de Karnaugh de F3

GRUPO 0		
cx.	elem.	tick
1	1,0	T
1	2,0	T
2	12,0	T
2	17,0	T
2	18,0	T
2	20,0	T
2	24,0	T
3	13,0	T
3	19,0	T
3	21,0	T
3	25,0	T
4	15,0	T
4	23,0	T
4	27,0	T
4	29,0	T
5	31,0	T

CUSTO = 72

GRUPO 1		
cx.	elem.	tick
1	1,16	F
1	2,16	F
2	12,1	F
2	17,2	T
2	17,4	T
2	17,8	T
2	18,1	F
2	20,1	F
2	24,1	F
3	13,2	T
3	13,16	T
3	19,4	T
3	19,8	T
3	21,2	T
3	21,8	T
3	25,2	T
3	25,4	T

GRUPO 1		
cx.	elem.	tick
4	15,16	T
4	23,8	T
4	27,4	T
4	29,2	T

GRUPO 2		
cx.	elem.	tick
2	17,6	T
2	17,10	T
2	17,12	T
3	13,18	F
3	19,12	T
3	21,10	T
3	25,6	T

GRUPO 3		
cx.	elem.	tick
2	17,14	F

Figura III.7.- Minimização de F3

Da figura III.7 obtém-se o conjunto dos implicantes-primos de F3 que é dado pelos termos não marcados (Tick = F):

(17,14) = vz
(13,18) = wxz
(24,1) = vwx'y'
(20,1) = vw'xy'
(18,1) = vw'x'y
(12,1) = v'wxy'
(2,16) = w'x'yz'
(1,16) = w'x'y'z

IMPLICANTES	mintermos de F3											
	13	15	17	18	19	20	21	23	25	27	29	31
(17,14)			X		X		X	X	X	X	X	X
(13,18)	X	X									X	X
(24, 1)									X			
(20, 1)						X	X					
(18, 1)				X	X							
(12, 1)	X											
(2,16)				X								
(1,16)			X									

CUSTO = 37

Figura III.8.- Mapa dos implicantes-primos de F3.

IMPLICANTES	18
(24, 1)	
(18, 1)	X
(12, 1)	
(2,16)	X
(1,16)	

Figura III.9.- Mapa reduzido dos implicantes-primos de F3.

Logo, para F3 são obtidas 2 expressões mínimas com custo=17.

F3 = (17,14) + (13,18) + (20,1) + (18,1)
= vz + wxz + vw'xy' + vw'x'y

ou

F3 = (17,14) + (13,18) + (20,1) + (2,16)
= vz + wxz + vw'xy' + w'x'yz'

Note a redução dos custos, inicialmente 72, com a obtenção dos implicantes primos passa a 37, e com a redução do mapa cai para 17.

III.4.- COBERTURA ENTRE IMPLICANTES PRIMOS E ENTRE MINTERMOS

Nas definições dos níveis de essenciais estão implícitas duas extensões do conceito de cobertura : "cobertura entre implicantes-primos" e "cobertura entre mintermos" (linhas e colunas do mapa, respectivamente).

Um implicante-primo A cobre um implicante-primo B (numa tabela reduzida ou quando existirem don't care states), se toda vez que B=1, também A=1, ou seja, A cobre qualquer mintermo que B cobrir no mapa. Na figura III.8, nota-se por exemplo que o implicante (17,14) cobre o implicante (24,1), o que permite retirar este último da tabela afim de reduzi-la.

Também se define que um mintermo i cobre um mintermo j (diz-se também que i domina j) se, no mapa dos implicantes primos, em cada linha onde houver uma marca na coluna j for encontrada também uma marca na coluna i. O importante aqui é ressaltar que para reduzir a tabela com esta propriedade, é eliminado o mintermo dominante (i), pois toda expressão mínima deduzida a partir do mintermo i pode ser obtida do mintermo j. Neste exemplo, na figura III.8, observa-se que o mintermo 25 domina o mintermo 23 pois em todas as linhas onde 23 possui uma marca, 25 também a tem.

Exemplo 4.- Minimizar a função F4 cujo custo inicial = 70.

F4(v,w,x,y,z) = S (0,1,3,4,7,13,15,19,20,22,23,29,31)

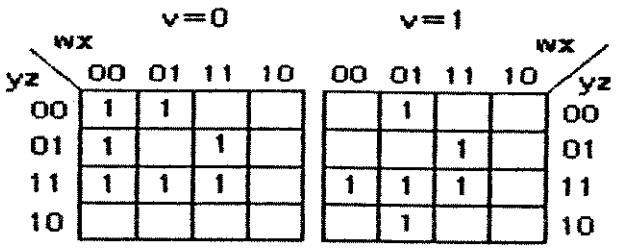


Figura III.10.- Mapa de Karnaugh de F4= S(0,1,3,4,7,13,15,19,20,22,23,29,31)

O mapa dos implicantes-primos de F4 obtido através do método tabular de Quine-McCluskey é dado na tabela da figura III.11 :

IMPLICANTES	mintermos de F4												
	0	1	3	4	7	13	15	19	20	22	23	29	31
(13,18)						X	X					X	X
(7,24)					X		X				X		X
(3,20)			X		X			X			X		
(22, 1)										X	X		
(20, 2)									X	X			
(4,16)				X					X				
(1, 2)		X	X										
(0, 4)	X			X									
(0, 1)	X	X											

CUSTO = 42

Figura III.11.- Mapa dos implicantes-primos de F4

Na figura III.11 as linhas assinaladas indicam os implicantes-primos essenciais e as colunas indicam os mintermos que caracterizaram os implicantes primos essenciais. Reduzindo-se esta tabela com a retirada destes essenciais e dos correspondentes mintermos, obtém-se a tabela da figura III.12:

IMPLICANTES	MINTERMOS				
	0	1	4	20	22
(22, 1)					X
(20, 2)				X	X
(4,16)			X	X	
(1, 2)		X			
(0, 4)	X		X		
(0, 1)	X	X			

Figura III.12.- Mapa reduzido de F4

Na figura 2.12, nota-se que não existe implicante primo essencial, mas observa-se que o implicante primo (20,2) cobre o implicante primo (22,1), e o (0,1) cobre o (1,2). Assim, pode-se retirar (22,1) e (1,2) da tabela, obtendo-se a tabela da figura III.13.

IMPLICANTES	MINTERMOS				
	0	1	4	20	22
(20, 2)				X	X
(4,16)			X	X	
(0, 4)	X		X		
(0, 1)	X	X			

Figura III.13.- Segunda redução do mapa de F4

Os implicantes (20,2) e (0,1) tornam-se essenciais e podem ser retirados

da tabela, juntamente com os mintermos cobertos por eles, 0, 1, 20 e 22, resultando na figura III.14 :

IMPLICANTES	4
(4,16)	X
(0, 4)	X

Figura III.14.- última redução do mapa de F4

Como (4,16) e (0,4) têm o mesmo custo, pode-se escolher qualquer um deles para cobrir o mintermo 4. Por exemplo, escolhe-se (0,4). Com isto, todos os mintermos estão cobertos e uma fórmula mínima da função é dada por :

$$F4 = (13,18) + (3,20) + (20,2) + (0,1) + (0,4)$$
$$= X11X1 + X0X11 + 101X0 + 0000X + 00X00$$
$$= wxz + w'yz + vw'xz' + v'w'x'y' + v'w'y'z'$$

CUSTO FINAL = 23

III.5.- MÉTODO DE RAMIFICAÇÃO

Na aplicação do algoritmo de Quine-McCluskey pode-se obter um mapa de implicantes-primos "cíclico", ou seja, não há nenhum implicante-primo essencial e nenhuma linha ou coluna pode ser eliminada da tabela pelos critérios de cobertura. Nestes casos utiliza-se o método de Ramificação descrito através do exemplo 5.

Exemplo 5.- Minimizando F5, figura III.15, é obtido o mapa de implicantes primos mostrado na figura III.16 :

$$F5 = S\ (0,1,5,7,8,10,14,15)$$

CUSTO INICIAL = 40

yz \ wx	00	01	11	10
00	1			1
01	1	1		
11		1	1	
10			1	1

F5

Figura III.15.- Mapa de Karnaugh para F5 = S (0,1,5,7,8,10,14,15)

Observa-se na figura III.16 que não há essenciais, nem linhas ou colunas dominantes. O método de Ramificação consiste em selecionar "arbitrariamente",

entre os de menor custo (menor número de literais), um implicante-primo como essencial.

IMPLICANTES	mintermos de F5							
	0	1	5	7	8	10	14	15
(0, 1)	X	X						
(1, 4)		X	X					
(5, 2)			X	X				
(7, 8)				X				X
(14, 1)							X	X
(10, 4)						X	X	
(8, 2)					X	X		
(0, 8)	X				X			

Figura III.16.- Mapa dos implicantes-primos de F5

Por exemplo, o mintermo 0 é coberto pelos implicantes (0,1) e (0,8). Como ambos possuem o mesmo número de literais, um deles é escolhido arbitrariamente e então, o mapa é reduzido pelas técnicas já citadas. Se for obtido novamente um mapa cíclico em qualquer passo, é escolhida novamente uma linha arbitrária e repetido o processo.

	MINTERMOS					
IMPLICANTES	1	5	7	10	14	15
(0, 1)	X					
(1, 4)	X	X				
(5, 2)		X	X			
(7, 8)			X			X
(14, 1)					X	X
(10, 4)				X	X	
(8, 2)				X		

Figura III.17.- Simplificação da figura III.16 pela retirada de (0,8), escolhido arbitrariamente como essencial, e dos mintermos 0 e 8, cobertos por ele .

Para a função F5 é selecionado o implicante (0,8) como essencial e a tabela da figura III.16 é reduzida à figura III.17, onde se pode identificar linhas dominadas, voltando a reduzi-la na figura III.18, onde identificam-se os implicantes primos essenciais (1,4) e (10,4) resultando na fig III.19.

	MINTERMOS					
IMPLICANTES	1	5	7	10	14	15
(1, 4)	X	X				
(5, 2)		X	X			
(7, 8)			X			X
(14, 1)					X	X
(10, 4)				X	X	

Figura III.18.- Redução da figura III.17

IMPLICANTES	7	15
(5, 2)	X	
(7, 8)	X	X
(14, 1)		X

Figura III.19.- Redução da figura III.18 pela retirada dos essenciais (1,4) e (10,4) e dos mintermos 1, 5, 10, 14, cobertos por eles.

Na figura III.19 , o implicante (7,8) cobre os demais implicantes, tornando-se essencial, e cobrindo os mintermos 7 e 15. Portanto,

$$F5 = (7,8) + (1,4) + (10,4) + (0,8) \qquad \text{(III.5)}$$

$$= X111 + 0X01 + 1X10 + X000$$

$$= xyz + w'y'z + wyz' + x'y'z'$$

CUSTO FINAL = 16

Neste exemplo pode-se notar novamente a redução gradativa dos custos. No início, representando F5 por todos os seus mintermos, o custo é 40. Representando-a por todos os implicantes primos o custo cai para 32 e determinando os essenciais o custo é reduzido para 16.

III.6.- ALGORITMO DE CARUSO [2]

O algoritmo de Caruso utiliza grafos com a estrutura de árvore para a obtenção dos implicantes primos e cobertura de uma função booleana.

Utilizando-se de uma seleção local de implicantes, procura contornar o problema da geração de todos os implicantes da função para a obtenção dos implicantes primos essenciais.

A idéia básica é localizar um mintermo de acordo com algum critério de custo e a partir dele obter a maior célula que o contém.

Sua principal característica é a eficiência na obtenção dos implicantes

primos, além de efetuar simultaneamente a cobertura da função.

A sua desvantagem é não garantir a obtenção da função de custo mínimo e sim uma cobertura irredundante.

III.6.1.- NOTAÇÃO E DEFINIÇÕES

A descrição do algoritmo de Caruso feita neste trabalho difere bastante da original. O autor acredita que a notação aqui utilizada é mais apropriada para a implementação computacional.

V(N) : é o conjunto dos decimais de todos os vértices do espaço binário N-dimensional; onde N é o número de variáveis da função.

F : é o conjunto dos decimais que representa todos os vértices do espaço binário N-dimensional, pertencentes a função; F está contido em V(N).

Peso : Peso de um bit é o valor decimal igual a $2^{(k-1)}$, onde K é a posição relativa do bit, contada a partir da menos significativa.

(u) : é o conjunto dos pesos binários das N variáveis de uma dada função.

v : é o decimal que representa um vértice da função (mintermo ou irrelevante).

m : é o decimal que representa um mintermo da função.

Distância : A distância Hamming entre dois números binários, é definida como o número de posições em que os números diferem.

m[ ]D1 : é um subconjunto ordenado de pesos, pertencentes a (u), que leva a vértices a distância 1 de m.

Operação invbit : Aplicar invbit(p), com p pertencente a (u), a um vértice v, significa complementar o bit de peso p da representação binária de v.

A notação utilizada :

$$v \xleftarrow{p} vx$$

onde vx é um vértice a distância 1 de v.

vx é um sucessor de v a distância 1 se pertencer a função.

Aplicar a operação invbit(4) ao vértice 24 significa inverter o valor do bit de peso 4 do vértice 24.

$$\underset{(11000)}{24} \xleftarrow{4} \underset{(11100)}{28}$$

peso 4 peso 4

Define-se ordem de um mintermo como sendo o número de elementos do

conjunto m[ ]D1.

m[ ] : é um subconjunto de m[ ]D1 com ordem k de m[ ]D1.

Aplicar m[ ] significa aplicar a operação invbit com todos os $2^k - 1$ subconjuntos de m[ ] ao mintermo m;

Grafo : é a representação gráfica das aplicações de m[ ];

Aplicar 11[8,4,1] significa criar um grafo aplicando a operação invbit(p) ao vértice 11, sendo p os subconjuntos de (8,4,1).

A figura III.20 apresenta o grafo criado pela aplicação de 11[8,4,1].

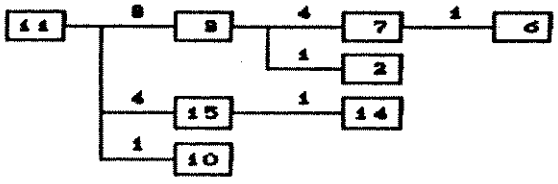


FIGURA III.20.- Aplicação de 11[8,4,1].

Define-se peso-referência ao decimal resultante da soma dos pesos de m[ ]D1. Deste modo 11[8,4,1] é representado por 11[13], onde 13 é o peso-referência.

O vértice 11 é denominado de raiz do grafo.

Conjunto irredundante de grafos representando uma função F é aquele em que todo mintermo da função está presente em pelo menos um grafo do conjunto, e em todos os grafos existe pelo menos um mintermo que não está em nenhum outro grafo deste conjunto.

Vértice singular é aquele que está somente em um grafo de um determinado conjunto irredundante.

m[ ] é um implicante (m[ ]I) cobrindo m se, e somente se, todos os vértices gerados pela aplicação de m[ ] pertencerem a função;

m[ ]I é um implicante primo (m[ ]P) se, e somente se, os vértices gerados não forem um subconjunto de nenhum outro m[ ]I para um dado m;

Exemplo 6. Considere a função $F6(A,B,C,D) = S\ (0,1,7,8,9,12,13)$

A figura III.21 apresenta o mapa de Karnaugh para F6.

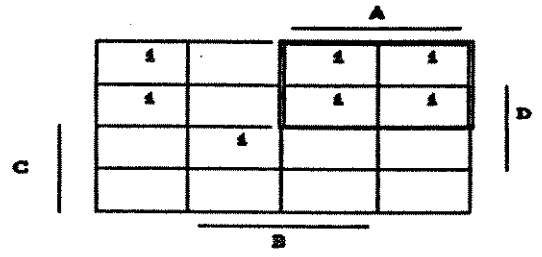
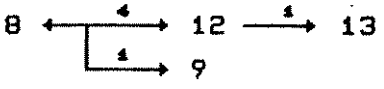


FIGURA III.21.- Mapa de Karnaugh para a função F6

8[4] é um implicante de F6 cobrindo 8 (8[4]I) pois da aplicação de 8[4] é obtido o vértice 12 pertencente a F6.

$$8 \xrightarrow{4} 12$$

8[5] é um implicante primo (8[5]P) de F6 pois da aplicação de 8[5] é obtido um implicante que não está contido em nenhum outro implicante.



8[4]I não é um implicante primo porque está contido em 8[5]P.

Para se obter implicantes primos a partir de um grafo, procede-se da seguinte maneira:

Identifique os vértices terminais de maior distância da raiz. Seus subconjuntos de pesos [A,B,...] formam os m[A]I, m[B]I, ... etc. Marcar no grafo todos os vértices obtidos pela aplicação de cada m[ ]P encontrado. Repetir este procedimento para os vértices terminais ainda não marcados. Parar quando todos os vértices terminais do grafo tiverem sido marcados.

Exemplo 7. Obtenha os implicantes primos da função definida pelos mintermos apresentados no grafo da figura III.22.

Note que o vértice 17 não é um vértice da função.

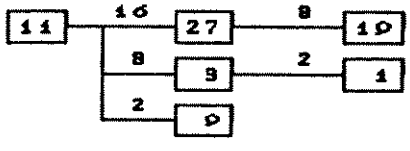
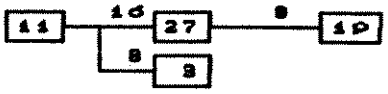


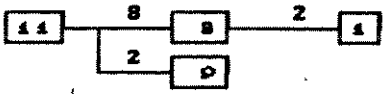
FIGURA III.22.- Grafo obtido pela aplicação de 11[26].

Os vértices 19 e 1 são os vértices terminais de maior distância da raiz.

Aplicando 11[24] obtém-se os vértices 27,19 e 3.



Aplicando 11[10] obtém-se os vértices 3,1 e 9



Do grafo apresentado na figura III.22 são extraídos os implicantes primos 11[24] e 11[10].

A figura III.23 apresenta os mintermos do grafo da figura III.22 no mapa de Karnaugh.

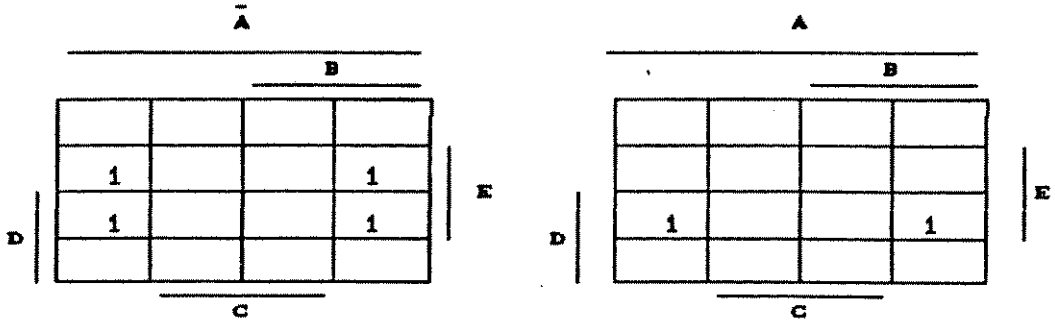


FIGURA.III.23.- Apresentação dos mintermos do grafo da figura III.22 no mapa de Karnaugh.

P(v) : é o conjunto de todos os implicantes primos que cobre um vértice da função.

No grafo da figura III.22 pode-se verificar que o vértice 3 é coberto pelos implicantes primos 11[24]P e 11[10]P, portanto P(3) é representado pelo conjunto (11[24]P, 11[10]P).

Se para um dado mintermo existir um único implicante primo, este implicante é chamado de implicante primo essencial (IPE).

Uma cobertura de um grafo é qualquer conjunto de implicantes primos extraído do grafo, que seja uma cobertura irredundante para todos os vértices singulares deste grafo. Isto é, cada implicante primo desta cobertura deve cobrir pelo menos um vértice singular que não é coberto por nenhum outro implicante primo da cobertura e, todos os vértices singulares do grafo devem ser cobertos.

Para escolher a cobertura de um grafo, deve-se identificar qual o tipo de grafo gerado.

Grafo completo: Neste grafo existe um único implicante primo no conjunto p[v] que cobre um dado vértice. Este vértice é chamado vértice IPE-ISOLADO e todos os vértices do grafo formam um implicante primo essencial (IPE).

Grafo Parcialmente Completo: Neste grafo existe pelo menos um implicante primo (IP1) no conjunto p[v] que satisfaz as seguintes condições:

A. Nenhum outro implicante primo (IP) pertencente a p[v] tem mais elementos que IP1 no conjunto (u).

B. Todos os mintermos cobertos por outro IP de p[v] podem ser cobertos por IP1. O vértice associado a IP1 é chamado vértice D-Isolado.

Grafo com menos de $2^k - 1$ vértices que possui cobertura única: Neste grafo o conjunto de todos os IP's essenciais à cobertura cobre todos os vértices singulares do grafo.

Além dos três tipos de grafos apresentados existe um quarto tipo que não apresenta nenhuma das características anteriormente citadas. A cobertura para este tipo de grafo exige a utilização de métodos clássicos de seleção de implicantes primos. Caruso [2] sugere que se escolha arbitrariamente o maior implicante primo do último grafo, pois a geração dos grafos está cronologicamente ordenada pelo custo decrescente de sua cobertura.

Cobertura irredundante da função é a união do menor número de implicantes primos suficiente para cobrir a função.

III.6.2.- DESCRIÇÃO DO ALGORITMO DE CARUSO

1. Ordene os mintermos, de forma crescente, de acordo com o número de seus sucessores à distância 1.

Os mintermos com o mesmo número de sucessores à distância 1 são ordenados de forma crescente pelos respectivos valores decimais.

2. Escolha o primeiro mintermo na função ordenada, gere o grafo correspondente de acordo com os passos A,B e C descritos a seguir e marque na função todos os vértices que aparecem neste grafo.

O mintermo escolhido é a raiz para o grafo gerado.

Gerar o grafo para uma raiz:

A. Aplicar todos os p pertencentes a m[ ]D1 ao mintermo m. Assim obtém-se todos os vértices vx com distância 1 de m.

B. Aplicar a cada vértice obtido todos os p pertencentes a m[ ]D1, subsequentes àqueles com o qual ele foi gerado.

C. Repetir o passo B para cada vértice obtido, até que não seja gerado mais nenhum vértice da função.

3. Verifique se o grafo é completo.

O teste consiste em verificar se na geração do grafo foi obtido algum vértice não pertencente a função. Em caso afirmativo o grafo não é completo.

Se o grafo é completo vá ao passo 5.

4. Verifique se o grafo é parcialmente completo.

O teste consiste em identificar os vértices irrelevantes de maior distância da raiz. Seus subconjuntos de pesos [A,B,...] formam os m[A]p, m[B]p,...,etc.

Marcar todos os vértices obtidos pela aplicação de cada m[ ]p encontrado. Se todos os mintermos do grafo forem marcados o grafo é parcialmente completo.

Se o grafo não é parcialmente completo vá ao passo 6.

5. Remova o grafo colocando-o na cobertura parcial da função. Sempre que remover um grafo, converta os seus mintermos em irrelevantes para a função e para os grafos restantes.

6. Se ainda existirem vértices não marcados na função, selecione o primeiro não marcado como a próxima raiz e gere o grafo correspondente. Note que a raiz é sempre um vértice singular.

A geração dos grafos até o passo 6 é repetida até que todos os mintermos da função estejam marcados.

7. Se restar algum grafo não removido vá ao passo 8. Caso contrário termine o algoritmo.

8. Marque com (s) todos os vértices singulares em todos os grafos que não foram removidos, com exceção dos vértices raiz.

9. Verifique para todos os grafos começando do primeiro se eles são parcialmente completos e se o conjunto de todos os implicantes primos essenciais à cobertura cobre todos os vértices singulares do grafo.

Em caso afirmativo proceda da mesma forma que no passo 5 e observe se aparecem novos vértices singulares.

Para que um grafo possa ser removido é necessário que todos os vértices singulares deste grafo sejam cobertos por pelo menos um implicante primo.

10. Repetir o passo 9 toda vez que pelo menos um grafo for removido, caso contrário vá ao passo 11.

11. Remover o último grafo gerado escolhendo sua cobertura de custo

mínimo e repetir o passo 8 se ainda existir grafo não removido. Caso contrário termine o algoritmo.

A cobertura da função é realizada removendo-se todos os grafos de um conjunto irredundante, um de cada vez.

Exemplo 8. Considere a função:

F7(A,B,C,D,E) = S (3,6,11,14,16,18,19,24,26,27,30).

A figura III.24 apresenta os grafos da aplicação da operação invbit(p) a todos os mintermos da função, sendo p o peso das variáveis de F7.

Pode-se observar que o mintermo 6 possui um único sucessor a distância 1; os mintermos 3,11,14,16,24 e 30 possuem 2 sucessores e o mintermo 26 possui 4 sucessores.

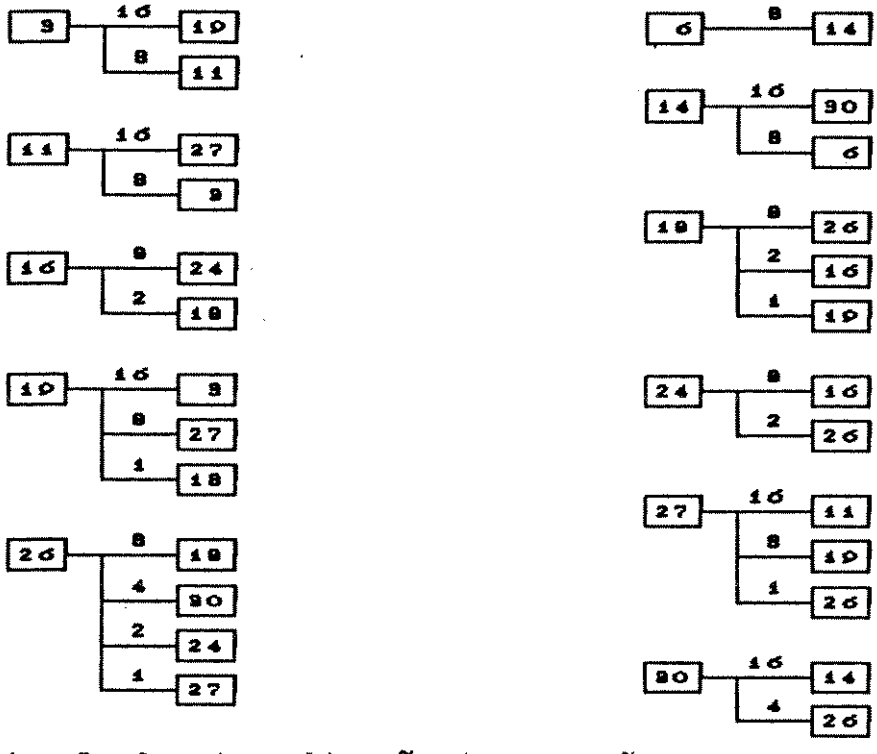
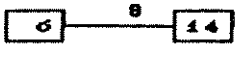


FIGURA III.24.- Grafos da aplicação da operação INVBIT(p) aos mintermos de F7.

Função ordenada: F7 = S (6,3,11,14,16,24,30,18,19,27,26).

Gere o grafo para a primeira raiz:



(6,8)

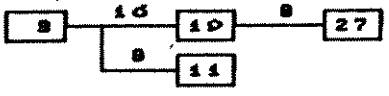
Este grafo é completo portanto deve ser removido.

Sempre que um grafo for removido os mintermos a ele associado devem ser transformado em irrelevantes na função e nos outros grafos.

A cobertura parcial da função é representada pelo implicante primo essencial (6,8).

$F7^4$ = S (3,11,16,24,30,18,19,27,26) + D(6,14)

Gere o grafo para a próxima raiz.



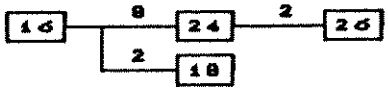
(3,24)

Grafo completo. O grafo é removido.

A cobertura parcial da função é representada pelos implicantes primos essenciais (6,8),(3,24).

$F7^2$ = S (16,24,30,18,26) + D(6,14,3,19,11,27)

Gere o grafo para a próxima raiz.



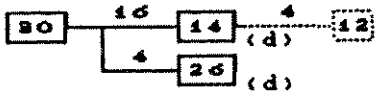
(16,10)

Grafo completo. O grafo é removido.

A cobertura parcial da função é representada pelos IPE (6,8),(3,24),(16,10).

$F7^9$ = S (30) + D(6,14,3,19,11,27,16,24,26,18)

Gere o grafo para a próxima raiz.



(14,16)

Grafo parcialmente completo pois na aplicação da operação invbit(4) ao vértice 14 foi gerado um vértice não pertencente a função e o grafo contém um IP1 que satisfaz as condições para o grafo parcialmente completo. O grafo é removido.

Note que poderia ter sido escolhido o implicante primo (26,4) para a cobertura deste grafo.

A cobertura parcial da função é representada pelos implicantes primos essenciais (6,8),(3,24),(16,10) e (14,16).

$F7^4$ = S () + D(6,14,3,19,11,27,16,24,26,18,30)

A função é expressa pela soma dos implicantes primos essenciais (cobertura irredundante) contidos na cobertura parcial da função.

A figura III.25.- Apresenta o mapa de Karnaugh para F7.

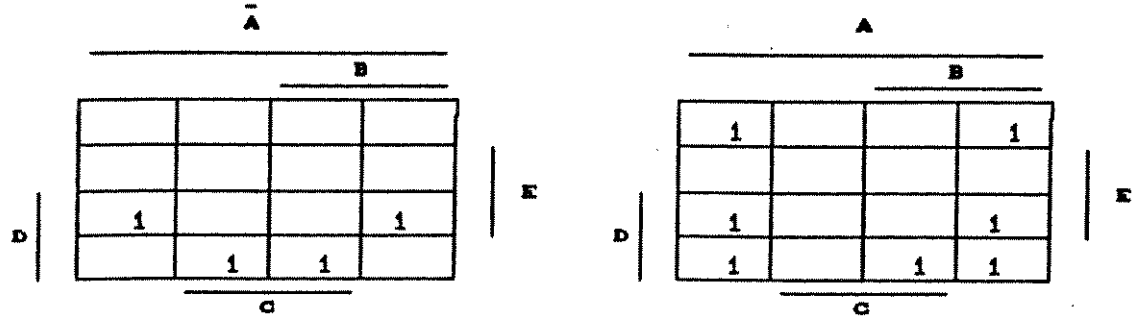


FIGURA III.25.- Mapa de Karnaugh para F7.

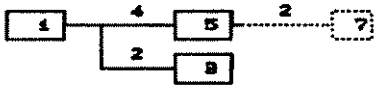
Exemplo 9. Considere a função F8 = S (1,2,3,4,5,6)

A figura III.26 apresenta os grafos da aplicação da operação invbit(p) a todos os mintermos de F8.

FIGURA III.26.- Grafo da aplicação da operação INVBIT(p) aos mintermos de F8.

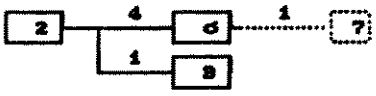
Função ordenada: F8 = S (1,2,3,4,5,6).

Gere o grafo para a primeira raiz:



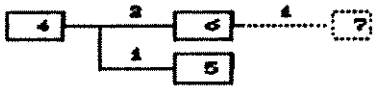
O grafo não é completo pois foi gerado um vértice que não pertence a função. Não é parcialmente completo, pois nenhum de seus vértices terminais é um irrelevante.

Gere o grafo para a próxima raiz.



O grafo não é completo e nem parcialmente completo.

Gere o grafo para a próxima raiz.



O grafo não é completo e nem parcialmente completo.

Todos os vértices da funcão já foram marcados e existem grafos que não foram removidos.

Deve-se aplicar o terceiro tipo de cobertura, ou seja marcar em todos os grafos os vértices singulares e verificar se o conjunto de todos os IP's essenciais à cobertura cobre todos os vértices singulares do grafo. Como os grafos não possuem vértices singulares o terceiro caso de cobertura falha.

Aplica-se então o quarto caso de cobertura que consiste em selecionar o maior implicante primo do último grafo. O grafo é removido e o implicante primo (4,2) vai para a cobertura parcial da função.

$$F8^{4} = S(1,2,3,5) + D(4,6)$$

Com a remoção do terceiro grafo, o vértice 5 do primeiro grafo passa a ser vértice singular.

Após a verificação do 3º caso de cobertura (parcialmente completo)o primeiro grafo é removido e o implicante primo (1,4) vai para a cobertura parcial da função.

$$F8^{2} = S(2,3) + D(4,6,1,5)$$

Com a remoção deste grafo, o vértice 3 do segundo grafo passa a ser vértice singular podendo-se remover o grafo e o implicante primo (2,1) vai para a cobertura parcial da função.

$$F8^{3} = S(\ ) + D(4,6,1,5,2,3)$$

A cobertura irredundante da função é a soma dos implicantes primos essenciais contidos na cobertura parcial da função.

$$F8 = (4,2) + (1,4) + (2,1)$$

A figura III.27 apresenta o mapa de Karnaugh para F8.

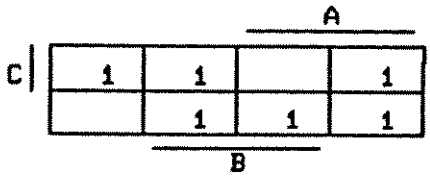


FIGURA III.27.- Mapa de Karnaugh para F8.

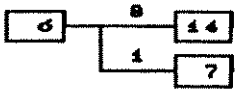
Exemplo 10: Considere a função :

F9(a,b,c,d) = S (0,1,3,6,7,8,11,14,16,18,19,24,26,27,30) + D(9)

Após encontrar os números de sucessores à distância 1 de cada vértice obtém-se a seguinte função ordenada.

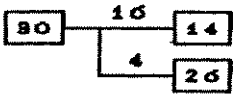
F9 = (6,7,14,30,0,1,8,11,16,18,19,24,27,3,26) + D(d)

Gere o grafo para a primeira raiz



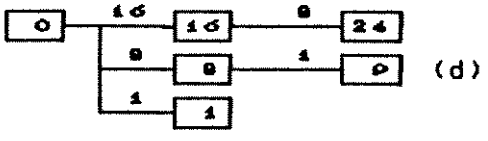
O grafo não é completo nem parcialmente completo.

Gere o grafo para a segunda raiz



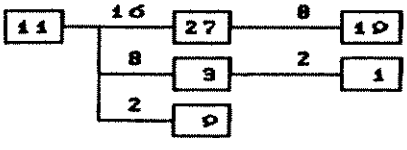
O grafo não é completo nem parcialmente completo.

Gere o grafo para a terceira raiz.



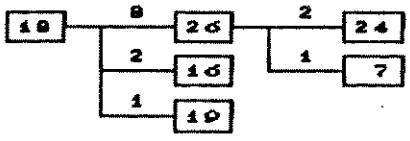
O grafo não é completo nem parcialmente completo.

Gere o grafo para a quarta raiz.



O grafo não é completo nem parcialmente completo.

Gere o grafo para a quinta raiz



O grafo não é completo nem parcialmente completo.

Verifica-se o terceiro caso de cobertura.

Marque com (s) todos os vértices singulares em todos os grafos.

O vértice 7 no primeiro grafo é singular pois não está presente em nenhum outro grafo. O vértice 8 no terceiro grafo e o vértice 3 no quarto grafo também são singulares.

O primeiro grafo é removido e gera o implicante primo (6,1). Com a remoção do primeiro grafo o vértice 14 do segundo grafo passa a ser singular. O grafo é removido e gera o implicante primo (14,16).

Com a remoção do segundo grafo o vértice 26 do quinto grafo passa a ser singular.

O terceiro grafo é removido e gera o implicante (0,24). Com a remoção deste grafo o vértice 1 do quarto grafo passa a ser singular.

O quarto grafo é removido e gera o implicante (1,10).

O quinto grafo é removido e gera o implicante (18,9).

A cobertura irredundante é a soma dos implicantes primos essenciais de todos os grafos.

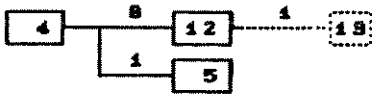
$$F9 = (6,1) + (14,16) + (0,24) + (1,10) + (18,19).$$

O algoritmo de Caruso não garante a função de custo mínimo e sim uma cobertura irredundante. O exemplo 11 é apresentado para mostrar um destes casos.

Exemplo 11. Considere a função $F10(A,B,C,D) = S(4,5,7,12,14,15)+D(3,8,9)$.

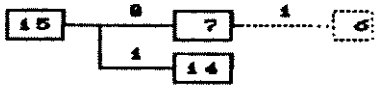
Função ordenada: $F10 = S(4,5,15,7,12,14) + D(3,8,9)$.

Gere o grafo para a primeira raiz.



O grafo não é completo e nem parcialmente completo.

Gere o grafo para a próxima raiz.



O grafo não é completo e nem parcialmente completo.

Verifique a possibilidade do terceiro caso de cobertura marcando nos grafos todos os vértices singulares. Pelos grafos pode-se verificar que os vértices 12,5,7 e 14 são vértices singulares.

Selecione para cada um dos grafos o implicante que cobre todos seus vértices singulares.

Utiliza-se o quarto método de cobertura, que consiste em escolher arbitrariamente os maiores implicantes primos que cobrem todos os vértices singulares do último grafo.

Os implicantes primos (7,8) e (14,1) são obtidos do segundo grafo.

Após a remoção deste grafo constata-se que novamente ocorre cobertura pelo 4º caso, isto é, os implicantes (4,8) e (4,1) são obtidos do primeiro grafo.

A cobertura irredundante é dada pelos implicantes primos essenciais (7,8),(14,1), (4,8) e (4.1).

Pelo mapa de Karnaugh apresentado na figura III.28 pode-se verificar que o algoritmo de Caruso não obteve a função de custo mínimo que é representada pelos implicantes (7,8),(12,2) e (4,1).

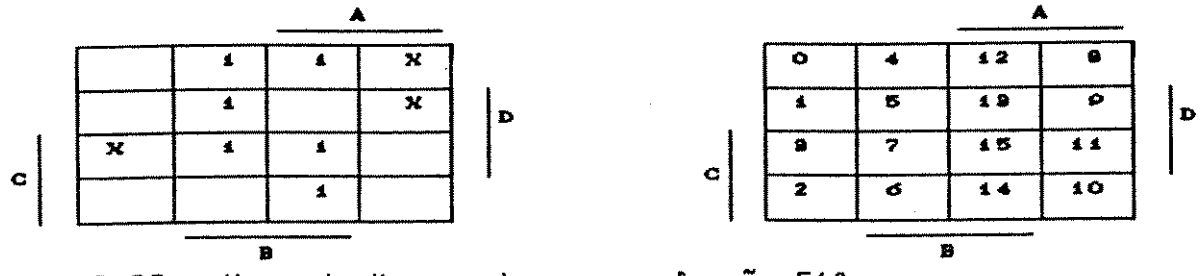


FIGURA III.28.- Mapa de Karnaugh para a função F10.

III.7.- ALGORITMO DE COBERTURA IRREDUNDANTE ATRAVÉS DA GERAÇÃO DE PADRÃO DE TESTE

A minimização através da Geração de Padrão de Teste determina uma cobertura irredundante de uma dada função booleana a partir de uma adaptação dos algoritmos inicialmente propostos para a detecção de falhas.

Dada uma cobertura inicial da função, através de falhas fixa-em-1 nas entradas das portas AND's e falhas fixa-em-0 na entrada da porta OR as possíveis redundâncias da cobertura inicial da função são detectadas e eliminadas.

O método proposto nesta seção tem como principal característica o uso restrito de memória e simplicidade de implementação, pois consiste apenas de duas listas ligadas: uma dos implicantes da função e os eventuais irrelevantes e outra da árvore de decisões dos testes.

Sua implementação computacional utiliza a alocação dinâmica de memória para armazenar os dados em uma estrutura de lista ligada, onde cada registro representa um implicante ou um irrelevante da função.

III.7.1.- CONCEITOS E DEFINIÇÕES

Nos circuitos lógicos, quando não existir teste para uma determinada falha esta é dita não detectável e o circuito é redundante com relação a falha. Este conceito é utilizado na simplificação das funções booleanas.

Uma expressão algébrica na forma canônica soma de produtos descrevendo a função pode ser representada por um circuito lógico composto de inversores, portas AND's e uma porta OR, onde as variáveis de cada mintermo da função são as entradas para as portas AND's correspondentes.

Por exemplo, a função $F(A,B) = AB + A\overline{B}$ equivale ao circuito lógico apresentado na figura III.29.

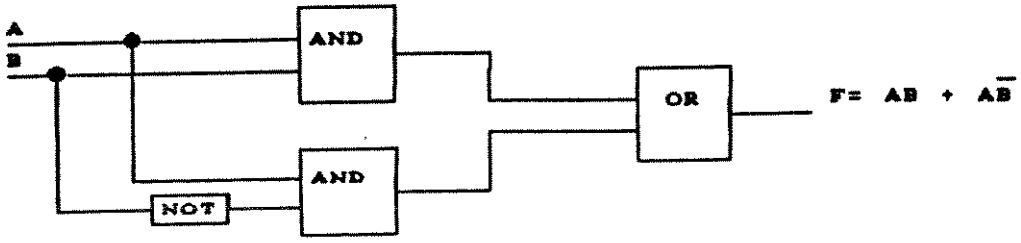


FIGURA III.29.- Circuito lógico representando a função booleana $F(A,B) = AB + A\bar{B}$.

Considere que uma falha f-e-1 em uma das entradas de uma porta AND seja não detectável. Isto significa que o circuito possui uma redundância em relação ao ponto testado. Deste modo a saída da porta AND não será alterada se se fixar permanentemente esta entrada com o valor lógico "1".

Uma porta AND com n entradas, com uma delas fixa em "1", é logicamente equivalente a uma porta AND com n-1 entradas.

Do mesmo modo uma falha f-e-0 não detectável na entrada de uma porta OR equivale a eliminar a porta AND a ela conectada que na minimização equivale a eliminar um implicante da função.

É importante salientar que uma falha f-e-0 na entrada de uma porta AND equivale a uma falha f-e-0 na entrada da porta OR ao longo deste caminho, de acordo com o conceito de propagação do sinal apresentado no capítulo II.

Todas as falhas detectáveis ao longo de um caminho sensibilizado são ditas equivalentes. Se um conjunto de falhas são equivalentes, qualquer teste que detecta uma delas irá detectar todas as outras. Deste modo, na geração de teste é necessário considerar somente uma falha de cada conjunto de falhas equivalentes.

Considere o circuito da figura III.30 e a falha f-e-0 no ponto

assinalado.

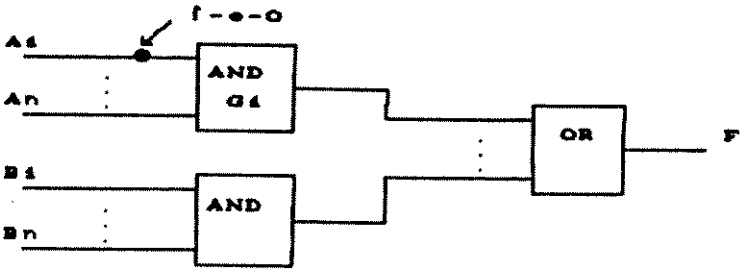


FIGURA III.30.- Circuito lógico apresentando uma falha f-e-0 na entrada A1 da porta G1.

Para que esta falha seja excitada é necessário que o sinal à esquerda do ponto assinalado esteja com o valor lógico 1 e para que o sinal seja propagado até a saída desta porta as outras entradas devem estar com o valor lógico 1. Nestas condições o cubo teste gerado corresponde a n-upla A1,A2,...,An = 1,1,...,1.

Para excitar uma falha f-e-0 no ponto assinalado da figura III.31 é necessário que o valor do sinal à esquerda deste ponto esteja em nível lógico 1 o que obriga que todas as entradas da porta G1 também estejam no nível lógico 1.

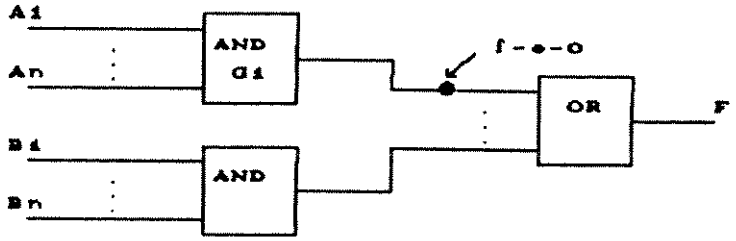


FIGURA III.31.- Circuito lógico apresentando uma falha f-e-0 no ponto assinalado.

Pode-se verificar que o cubo teste gerado para testar a falha f-e-0 no circuito da figura III.31 é o mesmo que foi gerado para testar a falha f-e-0 no circuito da figura III.30, o que mostra que uma falha f-e-0 na entrada da porta AND também testará a falha f-e-0 ao longo deste caminho até a saída do circuito.

O algoritmo de cobertura irredundante através da geração de padrões de teste utiliza-se da técnica de enumeração implícita na tentativa de encontrar um teste para a falha. Esta enumeração se faz através de uma pilha de teste que armazena as decisões nas variáveis e a porta que deu origem a decisão.

O algoritmo é bastante simples e utiliza-se de dois tipos de teste.

1.**Teste AND**: Consiste em testar uma falha f-e-1 na entrada de uma porta AND;

2.**Teste OR**: Consiste em testar uma falha f-e-0 na saída de uma porta AND o que corresponde a testar uma falha f-e-0 na entrada da porta OR.

Deste modo o algoritmo se resume em gerar testes para as falhas f-e-1 nas entradas das portas AND's e gerar testes para as falhas f-e-0 nas entradas da porta OR.

As falhas f-e-0 nas entradas das porta AND's são equivalentes à falha f-e-0 na entrada correspondente da porta OR e são tratadas no teste OR, enquanto que as falhas f-e-1 nas entradas da porta OR são irrelevantes para o métodos, motivo pelo qual não são tratadas.

Aplicar um teste significa verificar se a falha gerada pode ser propagada até a saída do circuito ou seja consiste na verificação da consistência do cubo teste.

Da aplicação de um teste quatro situações podem ocorrer:

A. Inconsistência no cubo teste (Conflito) :

Um conflito ocorre quando não for possível garantir a propagação do sinal até a saída do circuito, indicando que o cubo teste produz o valor lógico 1 na saída de alguma outra porta AND do circuito.

Neste caso a porta geradora do teste AND implica a porta AND onde o conflito foi gerado. Um conflito no teste AND, equivale também a um teste OR na saída da porta onde o conflito ocorreu permitindo eliminar-se a variável que gerou o teste e o elemento onde ocorreu o conflito.

B. Inconsistência no cubo teste possuindo apenas uma posição redundante (Implicação) :

A posição redundante deve ser alterada para garantir a propagação do sinal.

C. Inconsistência no cubo teste possuindo mais de uma posição redundante (Decisão).

Através da enumeração implícita as posições redundantes são alteradas (decisão) para garantir a propagação do sinal. Esta enumeração se faz através de uma pilha de teste que armazena a decisão na variável e a porta que deu origem a esta decisão.

D. Consistência no cubo teste: Neste caso o cubo teste difere em pelo menos uma das posições das variáveis em cada porta AND, o que garante a propagação da falha.

III.7.2.- Descrição do Algoritmo

1. Colocam-se os mintermos da função em uma fila de implicantes. Marca-se o início da fila à esquerda e o fim à direita.

2. Seleciona-se com um apontador o primeiro implicante da fila para efetuar o teste AND;

3. Efetua-se o teste AND em todas as suas variáveis sucessivamente.

O teste AND consiste em gerar um candidato a teste para uma falha tipo f-e-1 na posição da variável selecionada para teste. A variável é referenciada pelo peso que ela assume no mintermo, isto é, pelo valor de sua posição relativa. Assim a primeira variável à direita tem peso 0, a variável adjacente à sua esquerda tem peso 1, a próxima tem peso 2 e assim sucessivamente.

O cubo candidato a teste possui valor lógico complementado em relação ao implicante na posição da variável sob teste.

Após gerado o candidato a teste verifica-se sua consistência aplicando-o a todos os outros implicantes da fila. A análise de consistência é feita para se verificar se a falha se propaga para a saída do circuito. Nesta aplicação sistemática o cubo teste se modifica para garantir a consistência do teste. A determinação do teste se faz por enumeração implícita, isto é, a cada verificação de consistência pode-se tomar decisões, quanto ao valor das posições inicialmente redundantes que só garantirão uma inconsistência uma vez esgotada todas as possibilidades de teste.

Esta enumeração se faz através de uma pilha de teste que armazena as decisões nas variáveis e a porta que deu origem a decisão.

Uma vez detectada uma inconsistência elimina-se a variável sob teste, o que corresponde a expansão do implicante, e elimina-se o mintermo onde ocorreu a inconsistência, o que corresponde a uma cobertura.

4. Se no passo 3 houver eliminação de pelo menos uma variável, efetua-se o teste OR para o implicante resultante. Isto corresponde a cobertura de todos os mintermos do implicante. O teste OR é determinado nos mintermos à direita do apontador e a inconsistência é verificada no implicante em questão.

5. Se ainda existirem implicantes à direita do apontador avança-se o apontador e retorna-se ao passo 3. Caso contrário executa-se o passo 6.

6. Inserem-se os irrelevantes no final da fila e coloca-se o apontador no início da fila.

7. Repete-se o passo 3 apenas para os implicantes da fila. O passo 4 é aplicado apenas para os implicantes à direita do apontador e para os

irrelevantes, até que sejam testados todos os implicantes.

8. Retorna-se o apontador ao início da fila e marca-se o fim da fila no último implicante. Note que todos os implicantes presentes na fila são implicantes primos.

9. Aplica-se o teste OR final a todos os implicantes primos da fila. Em caso de inconsistência elimina-se o implicante primo que deu origem ao teste. Isto corresponde a reter na fila apenas os implicantes primos essenciais.

Os mintermos eliminados através do teste OR (passo 4), são justamente os mintermos onde ocorrem implicação no decorrer do teste AND (passo 3).

Por este motivo passa-se a marcar os mintermos onde ocorre implicação durante o teste AND, e no caso de inconsistência do teste, são eliminados além da variável sob teste os mintermos marcados. Isto possibilita a eliminação do teste OR intermediário (passo 4).

Somente mintermos à distância 1 podem ser agrupados. Isto leva a fazer uma pré-ordenação dos mintermos de acordo com o número de seus sucessores a distância 1, marcando nos mintermos as variáveis que devem ser testadas.

Estas modificações possibilitam a simplificação do algoritmo.

III.7.3.- Descrição do Algoritmo Simplificado

1. Colocam-se os mintermos da função em uma fila de implicantes, ordenada de forma crescente de acordo com o número de seus sucessores à distância 1.

Os mintermos com o mesmo número de sucessores a distância 1 são ordenados pelo número de sua representação decimal.

Marca-se o início da fila à esquerda e o fim à direita.

2. Anota-se para cada mintermo da fila as variáveis que tendo seu valor complementado levam a vértices a distância 1.

Os mintermos que não possuem sucessores a distância 1 (vértices isolados) são retirados da fila, pois não serão cobertos por nenhum outro implicante.

3. Seleciona-se com um apontador o primeiro implicante da fila para efetuar o teste AND.

O teste AND é gerado somente para as variáveis marcadas visto que elas levam a mintermos a distância 1.

Após gerado o cubo teste verifica-se sua consistência através da aplicação a todos os outros implicantes da fila.

A análise de consistência é feita para verificar se a falha se propaga para a saída do circuito. Esta operação é efetuada em duas etapas:

Na primeira etapa a aplicação é feita nos mintermos à direita do AND gerador do teste. Caso ocorra implicação deve-se marcar o mintermo implicado para que na ocorrência de um conflito possa ser eliminada a variável sob teste e o mintermo onde ocorreu o conflito (inconsistência do teste) assim como os mintermos marcados.

Isto corresponde a cobertura de todos os mintermos do implicante sob teste.

Na segunda etapa a aplicação é feita nos implicantes à esquerda do AND gerador do teste somente se na primeira etapa não ocorreu conflito. Na ocorrência de implicação o elemento implicado não deve ser marcado e no caso de conflito deve-se eliminar somente a variável sob teste.

Nesta aplicação sistemática o cubo teste se modifica para garantir a consistência do teste. A determinação do teste se faz por enumeração implícita, isto é, a cada verificação de consistência pode-se tomar decisões, quanto ao valor das posições inicialmente redundantes que só garantirão uma inconsistência uma vez esgotada todas as possibilidades de teste.

Uma vez detectada uma inconsistência elimina-se a variável sob teste, o que corresponde a expansão do implicante, e o mintermo onde ocorreu a inconsistência, o que corresponde a uma cobertura.

4. Se ainda existirem mintermos à direita do apontador avança-se o apontador e retorna-se ao passo 3. Caso contrário executa-se o passo 5.

5. Inserem-se os irrelevantes no final da fila e coloca-se o apontador no início da fila.

6. Repete-se o passo 3 apenas para os implicantes da fila.

7. Retorna-se o apontador ao início da fila e marca-se o fim da fila no último implicante. Note que todos os implicantes presentes na fila são implicantes primos.

8. Gere o teste OR para todos os implicantes da fila, aplicando-o aos demais implicantes primos.

Em caso de inconsistência elimina-se o implicante primo gerador do teste.

Isto corresponde a reter na fila apenas os implicantes primos essenciais.

9. A cobertura irredundante da função corresponde a todos os implicantes primos que restaram na fila e os eventuais mintermos que foram separados no início do algoritmo por não possuirem sucessores a distância 1.

Um exemplo da utilização do algoritmo de Cobertura Irredundante através da Geração de Padrão de Teste é apresentado detalhadamente.

Exemplo 12. Minimizar a função F11 = S(1,3,6,8) + D(10,11,12,13,14,15)

MINTERMOS INICIAIS :

Representação Ternária	Representação Cúbica
1000	(8,0)
0110	(6,0)
0011	(3,0)
0001	(1,0)

ESTADOS IRRELEVANTES :

1111	(15,0)
1110	(14,0)
1101	(13,0)
1100	(12,0)
1011	(11,0)
1010	(10,0)

APÓS A ORDENAÇÃO:

0001	(1,0)
0110	(6,0)
0011	(3,0)
1000	(8,0)

Seleciona-se o primeiro mintermo na fila ordenada.

GATE GERADOR DOS TESTES: 0001 (1,0)

Gere o teste AND para todas as variáveis marcadas neste mintermo iniciando pela mais significativa.

A única variável marcada é a de peso 1

VARIAVEL 1:

TESTE GERADO: 0011 (3,0)

Para gerar um teste basta inverter o valor lógico da variável sob teste.

O teste gerado deve ser comparado com todos os outros mintermos da fila.

G.COMPARADOS:

0110		(6,0)
0011	*Conflito.*	(3,0)

Ocorreu conflito pois o teste gerado coincide com os valores assumidos pelas

variáveis do mintermo 3.

Neste caso deve-se eliminar o mintermo onde ocorreu o conflito e a variável sob teste, o que corresponde em aumentar o implicante gerador do teste e eliminar o mintermo por ele coberto.

FILA APÓS SIMPLIFICACAO:

	00X1	(1,2)
	0110	(6,0)
	1000	(8,0)

Seleciona-se o próximo mintermo da fila para executar o teste AND.

GATE GERADOR DOS TESTES:	0110	(6,0)
	↑	
VARIAVEL 3:		
TESTE GERADO:	1110	(14,0)
G.COMPARADOS:		
	1000	(8,0)
	00X1	(1,2)

Observe que inicialmente o teste é comparado com os elementos à direita do gate gerador do teste, para depois ser comparado com os implicantes à esquerda

Seleciona-se o próximo mintermo da fila para gerar o teste AND.

GATE GERADOR DOS TESTES:	1000	(8,0)
	↑↑	
VARIAVEL 2:		
TESTE GERADO:	1100	(12,0)
G.COMPARADOS:		
	00X1	(1,2)
	0110	(6,0)
VARIAVEL 1:		
TESTE GERADO:	1010	(10,0)
G.COMPARADOS:		
	00X1	(1,2)
	0110	(6,0)

Após gerar o teste AND para todas as variáveis de todos os mintermos deve-se incluir na fila de implicantes os eventuais irrelevantes e novamente verificar o teste AND para as variáveis ainda marcadas de todos os implicantes da fila de implicantes.

GATE GERADOR DOS TESTES: 00X1 (1,2)

Para este implicante não foi gerado nenhum teste, pois a única variável, que tendo seu valor complementado leva a vértices a distância 1, já foi eliminada na fase anterior.

GATE GERADOR DOS TESTES: 0110 (6,0)
↑

VARIAVEL 3:

TESTE GERADO: 1110 (14,0)

G.COMPARADOS:

1000		(8,0)
1111		(15,0)
1110	*Conflito*	(14,0)

FILA APÓS SIMPLIFICACAO:

00X1	(1,2)
X110	(6,8)
1000	(8,0)
1111	(15,0)
1110	(14,0)
1101	(13,0)
1100	(12,0)
1011	(11,0)
1010	(10,0)

Seleciona-se o próximo implicante para o teste AND.

GATE GERADOR DOS TESTES: 1000 (8,0)
↑↑

VARIAVEL 2:

TESTE GERADO: 1100 (12,0)

G.COMPARADOS:

1111		(15,0)
1101		(13,0)
1100	*Conflito*	(12,0)

FILA APÓS SIMPLIFICACAO:

00X1	(1,2)
X110	(6,8)
1X00	(8,4)

```
                        1111                    (15,0)
                        1110                    (14,0)
                        1101                    (13,0)
                        1011                    (11,0)
                        1010                    (10,0)
VARIAVEL 1:
          TESTE GERADO: 1X10                    (10,4)
          G.COMPARADOS:
                        1111                    (15,0)
                        1101                    (13,0)
                        1011                    (11,0)
                        1010  Implicacão        (10,0)
```

Ocorreu uma implicação no cubo teste contendo uma posicão redundante (posicão que pode ser alterada para garantir a propagação da falha).
A posicão redundante do teste deve ser fixada com o valor oposto a posicão correspondente do elemento onde ocorreu a implicacão.

```
   TESTE APOS ALTERACAO: 1110                   (14,0)
         G. Comparados:
                        00X1                    (1,2)
                        X110  Conflito          (6,8)
```

O conflito ocorreu num elemento à esquerda do implicante gerador do teste devendo ser eliminada somente a variável geradora do teste.

```
     FILA APóS SIMPLIFICAÇÃO:
                        00X1                    (1,2)
                        X110                    (6,8)
                        1XX0                    (8,6)
                        1111                    (15,0)
                        1110                    (14,0)
                        1101                    (13,0)
                        1011                    (11,0)
                        1010                    (10,0)
```

Após todos os implicantes da fila de implicantes terem sidos testado verifica-se a cobertura deste implicantes, gerando o teste OR para cada um deles e comparando com os outros implicantes da fila.

```
COBERTURA DOS IMPLICANTES:
GATE GERADOR DOS TESTES:   00X1                        (1,2)
          G.COMPARADOS:
                           X110                        (6,2)
                           1XX0                        (8,6)

GATE GERADOR DOS TESTES:   X110                        (6,8)
          G.COMPARADOS:
                           1XX0     Implicacão         (8,6)

   TESTE APOS ALTERACAO:   0110                        (6,0)
          G.COMPARADOS:
                           00X1                        (1,2)
GATE GERADOR DOS TESTES:   1XX0                        (8,6)
          G.COMPARADOS:
                           00X1                        (1,2)
                           X110     Decisão            (6,8)
```

Ocorreu uma decisão no cubo teste porém com duas possibilidades de alteracão para garantir a propagacão. Optou-se por fixar a posicão menos significativa com o valor oposto a da mesma posicão do implicante que gerou o conflito.

```
   TESTE APÓS ALTERAÇÃO:   1X00                        (8,4)

   DECISAO TOMADA :
                    TESTE :  (8,6)
                    CONFLITO :  (6,8)
                    BIT ALTERADO :  1
```

O cubo teste anterior, a decisão, o elemento onde ocorreu o conflito e o bit do cubo teste alterado são inseridos na pilha de decisões para que no caso de um conflito possa-se retornar a este ponto e tomar uma outra decisão que garanta a propagacão da falha. Após verificar a cobertura para todos os implicantes, restará na fila de implicantes somente os implicantes primos essenciais da função.

```
COBERTURA IRREDUNDANTE

                    00X1                (1,2)
                    X110                (6,8)
                    1XX0                (8,6)

CUSTO INICIAL DA FUNCAO  -> 20
CUSTO ANTES DA COBERTURA -> 11
CUSTO FINAL DA FUNCAO    -> 11
MEMóRIA UTILIZADA  : 192.00 bytes
DURAÇÃO :              3.41 segundos
```

A figura III.31.a mostra o mapa de Karnaugh da função F11 onde pode-se constatar que os implicantes 00X1, X110 e 1XX0 constituem-se numa cobertura irredundante da função.

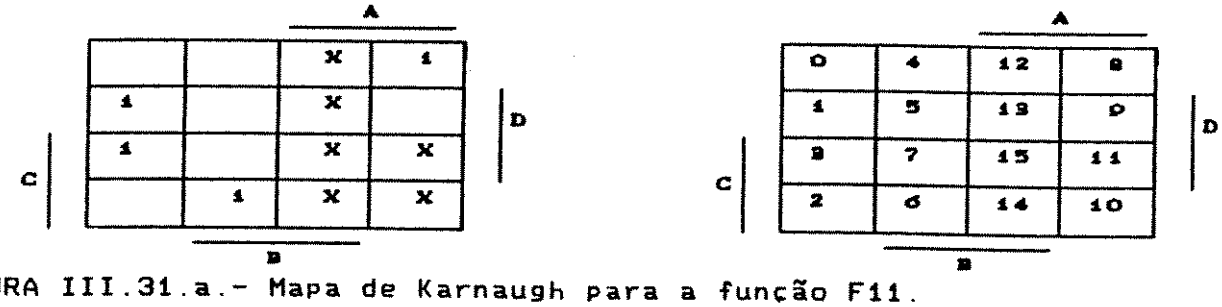


FIGURA III.31.a.- Mapa de Karnaugh para a função F11.

III.7.4.- DESCRIÇÃO DO PROGRAMA

Desenvolveu-se um programa de computador para IBM-PC XT ou AT na linguagem TURBO PASCAL versão 3 que minimiza funções booleanas de acordo com o algoritmo "Geração".

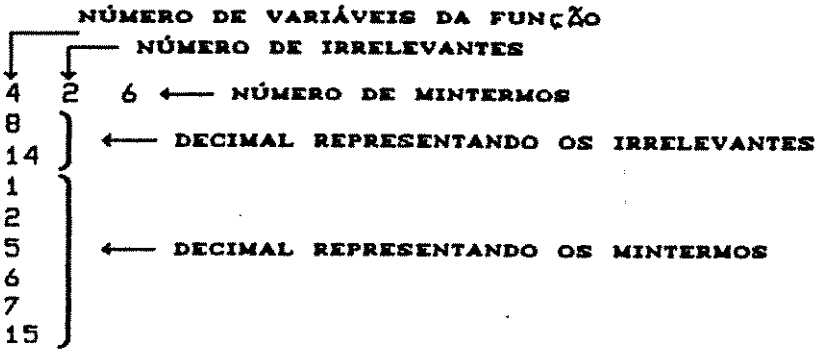
Para a utilização do programa é necessário fornecer os seguintes dados:

- Nome do arquivo que contém os mintermos e os irrelevantes da função;
- Nome do arquivo de saída dos resultados.

Por exemplo para descrever a função

F12(A,B,C,D) = S (1,2,5,6,7,15) + D(8,4)

é necessário criar um arquivo com as seguintes linhas:



As figuras III.32, III.33 e III.34 mostram os diagramas de blocos do programa que consiste de três fase:

- Teste AND nas variáveis dos mintermos;
- Teste AND nas variáveis dos implicantes (caso haja irrelevantes);
- Teste OR nos implicantes da função.

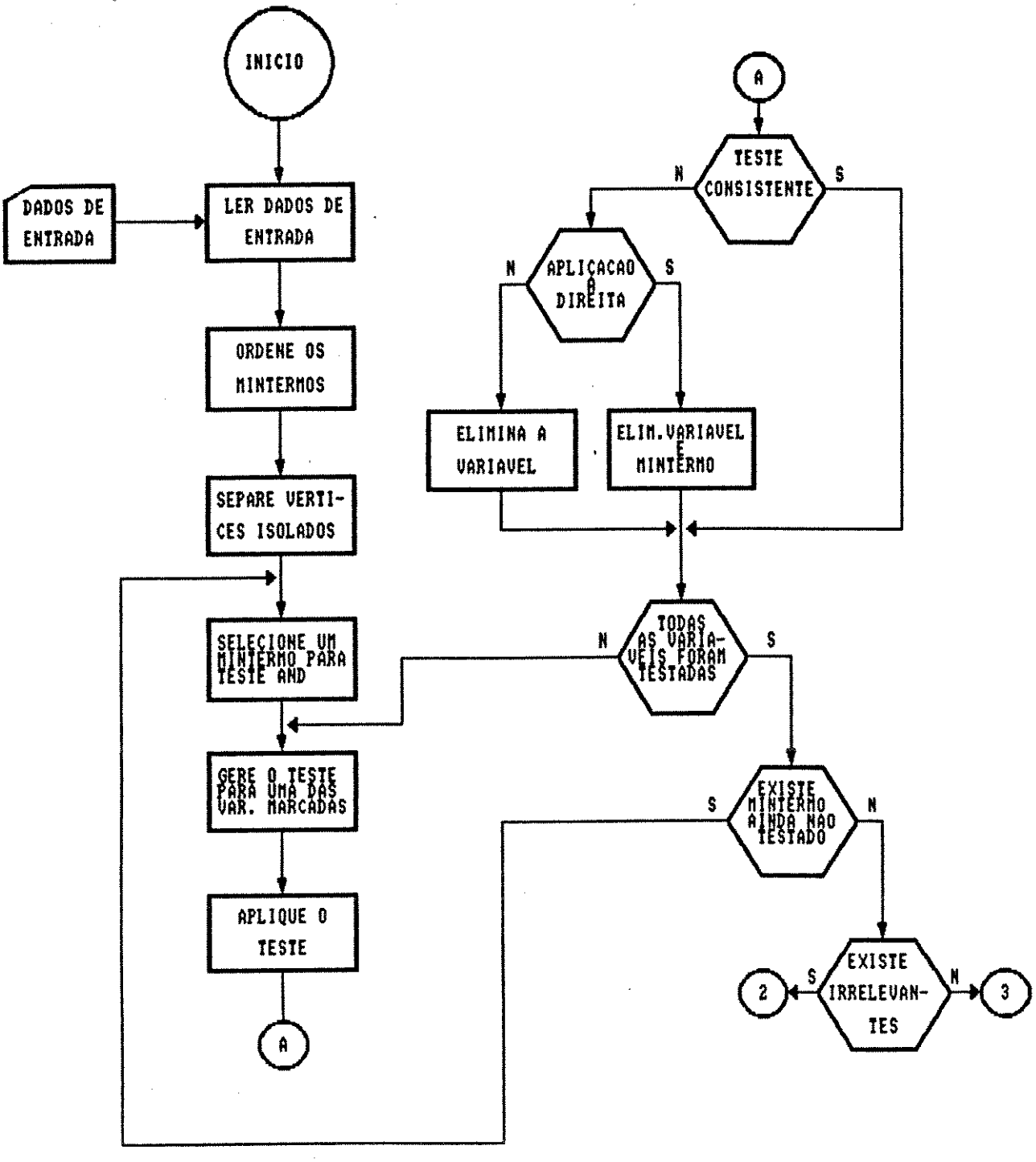


FIGURA III.32.- Diagrama de bloco da primeira fase do algoritmo

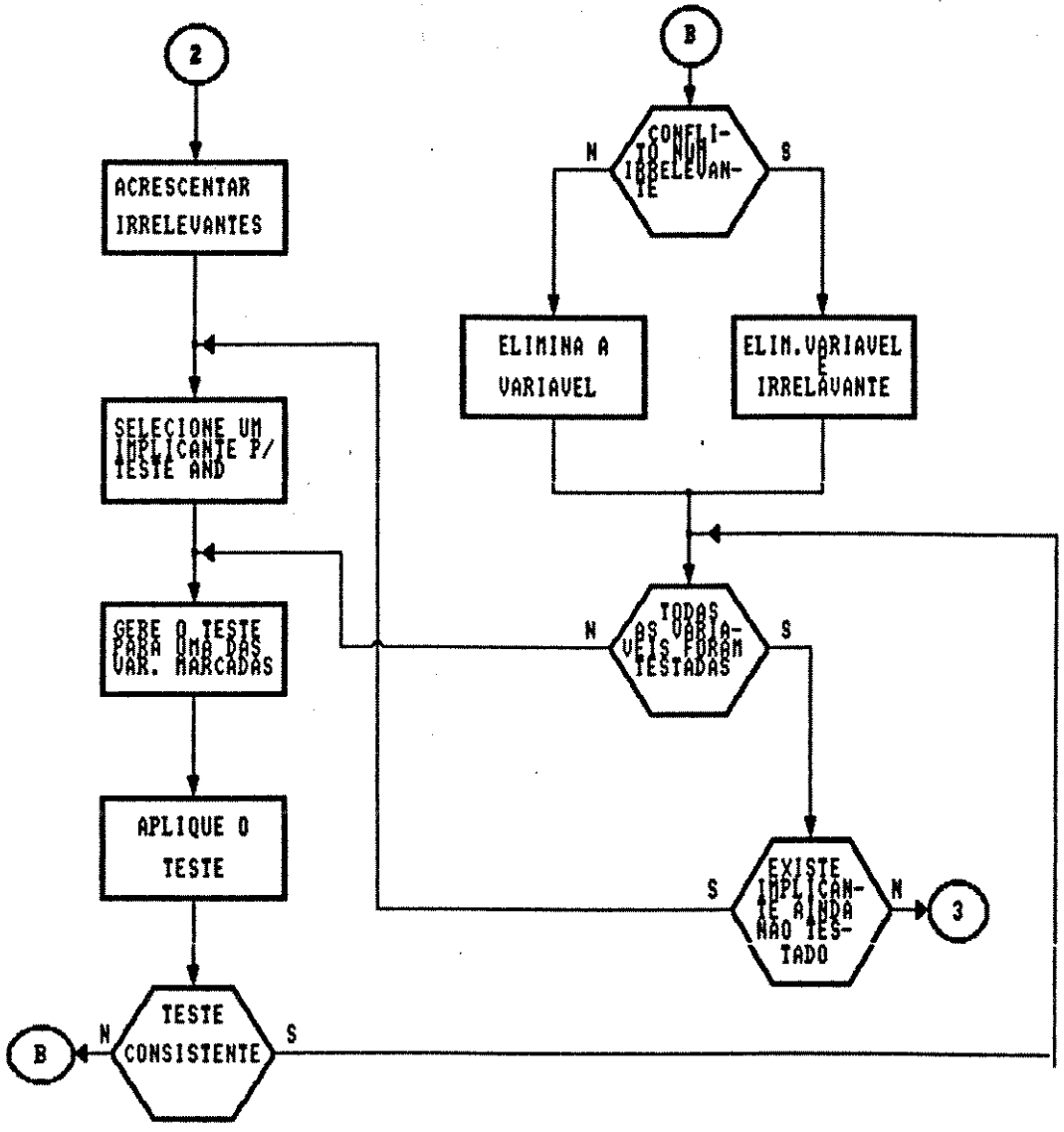


FIGURA III.33.- Diagrama de bloco da segunda fase do algoritmo

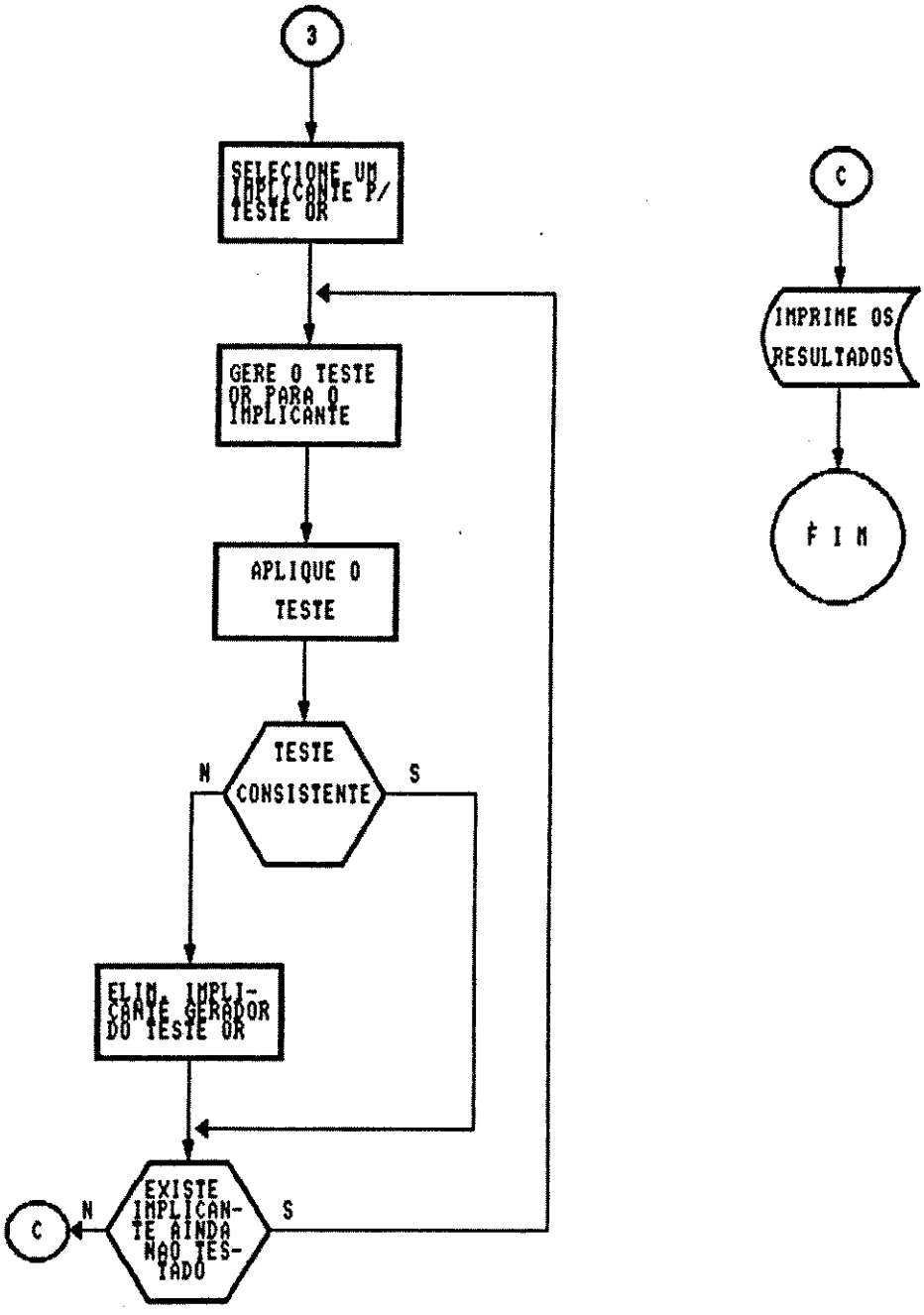


FIGURA III.34.- Diagrama de bloco da terceira fase do algoritmo

III.8.- COMPARAÇÃO DOS MÉTODOS

Com o propósito de validar o algoritmo "Geração", dezenas de casos de funções booleanas foram simplificadas e comparadas com os resultados obtidos pelos algoritmos de Quine-McCluskey e de Caruso.

O critério de custo adotado é o definido na secção II.3.2.

Para a comparação numérica foram selecionadas funções com N (número de variáveis) de 2 a 7.

Tabela com os casos rodados:

3.2.1.1 - F(A,B,C) = S (3,7) + D(5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.28	176	2
CARUSO	0.11	96	2
GERAÇÃO	0.11	64	2

3.2.2.1 - F(A,B,C) = S (3,7) + D(1,5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	240	1
CARUSO	0.16	144	1
GERAÇÃO	0.16	80	1

3.2.3.1 - F(A,B,C) = S (4,5) + D(0,6,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	288	1
CARUSO	0.22	160	1
GERAÇÃO	0.22	80	1

3.3.0.1 - F(A,B,C) = S (1,0,3,5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.28	176	6
CARUSO	0.27	128	6
GERAÇÃO	0.28	80	6

3.3.1.1 - F(A,B,C) = S (4,6,7) + D(5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.27	240	1
CARUSO	0.17	144	1
GERAÇÃO	0.27	80	1

3.4.0.1 - F(A,B,C) = S (1,6,3,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.27	224	6
CARUSO	0.27	144	6
GERAÇÃO	0.22	96	6

3.4.0.2 - F(A,B,C) = S (1,3,5,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.27	240	1
CARUSO	0.22	144	1
GERAÇÃO	0.16	80	1

3.4.0.3 - F(A,B,C) = S (4,3,5,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.22	224	6
CARUSO	0.27	144	6
GERAÇÃO	0.49	96	6

3.4.0.4 - F(A,B,C) = S (5,7,4,3)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.27	224	6
CARUSO	0.28	144	6
GERAÇÃO	0.27	96	6

3.4.0.5 - F(A,B,C) = S (0,1,4,6)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	224	6
CARUSO	0.28	144	6
GERAÇÃO	0.28	96	6

3.4.1.1 - F(A,C,C) = S (0,4,6,7) + D(5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	288	5
CARUSO	0.33	192	5
GERAÇÃO	0.71	96	5

3.4.2.1 - F(A,B,C) = S (4,5,3,7) + D(1,6)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	352	4
CARUSO	0.33	240	4
GERAÇÃO	0.55	112	4

3.5.0.1 - F(A,B,C) = S (6,7,3,4,5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	288	5
CARUSO	0.33	192	5
GERAÇÃO	0.38	112	5

3.5.0.2 - F(A,B,C) = S (0,1,2,3,5)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	288	5
CARUSO	0.39	192	5
GERAÇÃO	0.44	112	5

3.5.0.3 - F(A,B,C) = S (0,1,2,5,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	272	9
CARUSO	0.49	256	9
GERAÇÃO	0.55	112	9

4.2.6.1 - F(A,B,C,D) = S (4,9) + D(10,11,12,13,14,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	672	7
CARUSO	0.33	240	7
GERAÇÃO	1.09	112	7

4.4.0.1 - F(A,B,C,D) = S (1,5,9,13)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.28	240	2
CARUSO	0.22	144	2
GERAÇÃO	0.17	96	2

4.4.0.2 - F(A,B,C,D) = S (0,2,8,10)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.27	240	2
CARUSO	0.22	144	2
GERAÇÃO	0.22	96	2

4.4.0.3 - F(A,B,C,D) = S (4,6,12,14)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	240	2
CARUSO	0.28	144	2
GERAÇÃO	0.22	80	2

4.4.6.1 - F(A,B,C,D) = S (2,3,7,8) + D(15,14,11,12,13,10)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.38	592	0
CARUSO	0.71	368	0
GERAÇÃO	2.14	192	0

4.4.6.2 - F(A,B,C,D) = S (1,3,6,8) + D(10,11,12,13,14,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	560	11
CARUSO	0.61	304	11
GERAÇÃO	2.08	160	11

4.5.0.1 - F(A,B,C,D) = S (5,7,10,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.28	272	8
CARUSO	0.49	176	8
GERAÇÃO	0.38	112	8

4.5.0.2 - F(A,B,C,D) = S (0,2,8,12,13)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.32	272	12
CARUSO	0.55	256	12
GERAÇÃO	0.82	112	12

4.5.0.3 - F(A,B,C,D) = S (2,3,8,10,12)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	272	12
CARUSO	0.55	256	12
GERAÇÃO	0.77	112	12

4.5.3.1 - F(A,B,C,D) = S (0,7,8,10,12) + D(2,6,11)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	432	11
CARUSO	0.55	272	11
GERAÇÃO	2.15	144	11

4.5.6.1 - F(A,B,C,D) = S (5,6,7,8,9) + D(10,11,12,13,14,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	672	8
CARUSO	0.82	448	8
GERAÇÃO	2.59	192	8

4.6.0.1 - F(A,B,C,D) = S (11,7,13,15,5,3)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	352	6
CARUSO	0.49	240	6
GERAÇÃO	0.55	128	6

4.6.0.2 - F(A,B,C,D) = S (5,6,9,10,13,14)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	304	16
CARUSO	0.66	240	16
GERAÇÃO	1.26	128	16

4.6.0.3 - F(A,B,C,D) = S (0,8,12,14,5,7)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	304	12
CARUSO	0.50	208	12
GERAÇÃO	0.33	128	12

4.6.0.4 - F(A,B,C,D) = S (0,1,2,3,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	320	7
CARUSO	0.43	208	7
GERAÇÃO	0.66	112	7

4.6.0.5 - F(A,B,C,D) = S (0,2,10,11,12,14)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	320	12
CARUSO	0.49	208	12
GERAÇÃO	1.97	128	12

4.6.0.6 - F(A,B,C,D) = S (0,4,6,10,11,13)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	288	17
CARUSO	0.66	244	17
GERAÇÃO	1.32	128	17

4.7.0.1 - F(A,B,C,D) = S (0,1,2,4,5,8,10)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	400	6
CARUSO	0.55	256	6
GERAÇÃO	0.71	144	6

4.8.0.2 - F(A,B,C,D) = S (1,5,6,7,11,12,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	432	16
CARUSO	0.77	272	16
GERAÇÃO	2.15	176	16

4.8.0.3 - F(A,B,C,D) = S (12,0,5,9,3,15,6,10)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	0	40
CARUSO	1.92	272	40
GERAÇÃO	4.45	0	40

4.8.0.4 - F(A,B,C,D) = S (3,4,5,7,11,12,14,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.38	448	11
CARUSO	0.94	384	11
GERAÇÃO	1.48	160	11

4.9.0.1 - F(A,B,C,D) = S (0,1,3,6,9,11,12,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.38	480	19
CARUSO	1.21	368	19
GERAÇÃO	2.36	176	19

4.10.0.1 - F(A,B,C,D) = S (0,1,2,5,7,8,9,10,13,15) → Xadrez

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.38	576	9
CARUSO	1.15	480	9
GERAÇÃO	1.27	192	9

4.10.0.2 - F(A,B,C,D) = S (0,2,3,4,5,7,8,9,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.49	560	15
CARUSO	1.32	480	15
GERAÇÃO	3.07	192	29

4.10.0.3 - F(A,B,C,D) = S (0,1,2,5,7,8,9,10,13,15)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	576	0
CARUSO	1.15	480	0
GERAÇÃO	1.26	176	0

4.16.0.1 - F(A,B,C,D) = S (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) → Tudo um

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.82	1216	0
CARUSO	1.15	528	0
GERAÇÃO	2.07	336	0

5.6.0.1 - F(A,B,C,D,E) = S (9,20,21,29,30,31)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.33	304	21
CARUSO	0.88	288	21
GERAÇÃO	2.47	128	21

5.8.4.1 - F(A,B,C,D,E) = S (1,4,6,10,20,22,24,26) + D(0,11,16,27)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.28	656	18
CARUSO	1.21	464	18
GERAÇÃO	4.28	192	18

5.10.0.1 - F(A,B,C,D,E) = S (1,2,3,4,5,11,18,19,20,21)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	544	18
CARUSO	1.26	400	18
GERAÇÃO	2.36	192	18

5.11.0.1 - F(A,B,C,D,E) = S (3,6,11,14,16,18,19,24,26,27,30)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	608	18
CARUSO	1.48	448	18
GERAÇÃO	2.86	208	18

5.12.0.1 - F(A,B,C,D,E) = S (13,15,17,18,19,20,21,23,25,27,29,31)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	704	17
CARUSO	1.48	464	17
GERAÇÃO	2.12	240	17

5.14.0.1 - F(A,B,C,D,E) = S (0,1,4,5,6,11,12,14,16,20,22,28,30,31)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.48	800	22
CARUSO	2.14	544	22
GERAÇÃO	4.20	272	22

5.14.0.2 - F(A,B,C,D,E) = S (0,2,3,4,5,11,18,19,20,23,24,28,29,31)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.55	736	30
CARUSO	2.80	688	30
GERAÇÃO	8.74	256	30

5.15.1.1 - F(A,B,C,D,E) = S (0,1,3,6,7,8,11,14,16,18,19,24,26,26,30)+D(9)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	10.49	928	22
CARUSO	5.54	1696	22
GERAÇÃO	10.05	304	21

5.17.0.1 - F(A,B,C,D,E) = S (2,3,7,10,11,15,18,19,23,24,25,26,27,28,29, 30,31)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.66	1088	0
CARUSO	2.26	672	0
GERAÇÃO	3.90	384	0

5.17.0.2 - F(A,B,C,D,E) = S (0,1,3,4,5,7,8,9,10,12,13,21,24,25,26,28,29)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.60	1040	18
CARUSO	2.69	736	18
GERAÇÃO	5.11	320	18

5.24.0.1 - F(A,B,C,D,E) = S (0,1,2,4,5,7,8,10,11,13,14,15,17,18,19,20, 22,23,24,25,27,28,29,30)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	19.17	1456	32
CARUSO	11.48	2912	32
GERAÇÃO	20.21	416	56

6.11.7.1 - F(A,B,C,D,E,F) = S (0,2,14,18,21,27,32,41,49,53,62) + D(6,9,25,34,55,57,61)

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.44	912	46
CARUSO	3.02	640	46
GERAÇÃO	19.01	272	46

6.16.0.1 - $F(A,B,C,D,E,F) = S\ (2,3,6,7,10,14,18,19,22,23,27,37,42,43,45,46)$

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	0.55	896	27
CARUSO	2.42	560	27
GERAÇÃO	7.74	288	27

6.31.0.1 - $F(A,B,C,D,E,F) = S(1,2,3,4,5,8,9,10,17,20,21,24,25,27,32,33,34,36,37,40,41,42,43,44,45,46,47,48,56,59,62)$

	Tempo (seg)	Memória (bytes)	Custo
McCLUSKEY	2.79	1904	51
CARUSO	10.76	1840	51
GERAÇÃO	30.05	528	57

Resultados obtidos para os casos Tudo Um.

NUM VAR	Quine-McCluskey			CARUSO			GERAÇÃO		
	Memo	Tempo	Custo	Memo	Tempo	Custo	Memo	Tempo	Custo
2	240	0.29	0	144	0.17	0	96	0.05	0
3	496	0.33	0	272	0.33	0	160	0.34	0
4	1216	0.82	0	528	0.94	0	288	1.37	0
5	3136	4.84	0	1040	3.20	0	544	5.16	0
6	8000	40.21	0	2064	11.59	0	1056	16.90	0
7	21824	270.41	0	4112	45.91	0	2080	57.30	0

Os casos para as funções com mintermos múltiplos de N ou N-1 estão abaixo especificados.

$F(A,B,C) = S\ (2,3,4,6)$

$F(A,B,C,D) = S\ (3,4,6,8,9,12,15)$

$F(A,B,C,D,E) = S\ (4,5,8,10,12,15,16,20,24,25,28,30)$

$F(A,B,C,D,E,F) = S\ (5,6,10,12,15,18,20,24,25,30,35,36,40,42,45,48,50,54,55,60)$

F(A,B,C,D,E,F,G) = S (6,7,12,14,18,21,24,28,30,35,36,42,48,49,54,56,60,
63,66,70,72,77,78,84,90,91,96,98,102,105,108,112,
114,119,120,126)

F(A,B,C,D,E,F,G,H) = S (7,8,14,16,21,24,28,32,35,40,42,48,49,56,63,64,
70,72,77,80,84,88,91,96,98,104,105,112,119,120,
126,128,133,136,140,144,147,152,154,160,161,168,
175,176,182,184,189,192,196,200,203,208,210,216,
217,224,231,232,238,240,245,248,252)

Resultados comparativos para os casos múltiplos N e N-1.

NUM VAR	Quine-McCluskey			CARUSO			GERAÇÃO		
	Memo	Tempo	Custo	Memo	Tempo	Custo	Memo	Tempo	Custo
2	176	0.27	4	128	0.22	4	80	0.17	4
3	224	0.28	6	144	0.27	6	90	0.49	6
4	336	0.28	22	220	0.98	22	144	2.09	22
5	640	0.44	34	672	2.14	30	240	6.65	34
6	800	0.49	106	688	5.00	106	352	24.44	106
7	1632	1.04	160	1296	13.95	160	608	93.49	160

Para cada N (número de variáveis) foram geradas as funções Tudo Um, isto é, funções contendo todos os mintermos possíveis. Estas funções foram consideradas pois exigem grande tempo de CPU e espaço de memória.

Também foram geradas para cada N as funções cujos mintermos são os decimais múltiplos de N ou N-1 e funções cujos mintermos são os decimais miltiplos de N e N-2 e os irrelevantes os decimais múltiplos de N-1.

As figuras III.35 a III.48 mostram os resultados obtidos pelos diferentes métodos considerando-se o tempo de execução e a quantidade de memória utilizadas para cada método.

Note que o algoritmo "Geração" necessita menos espaço de memória que os dois outros.

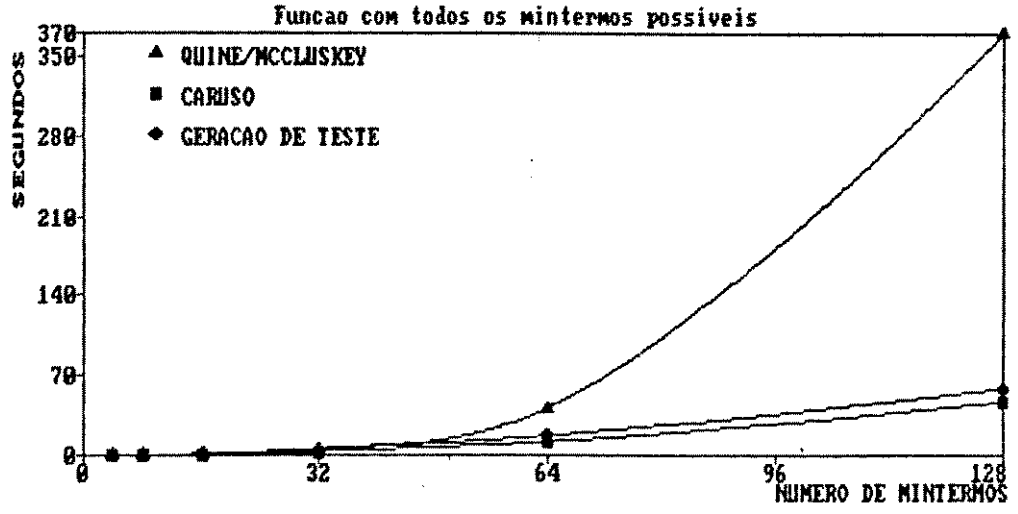


Figura III.35.- Comparação dos tempos de execução da simplificação das funções que contém todos os mintermos.

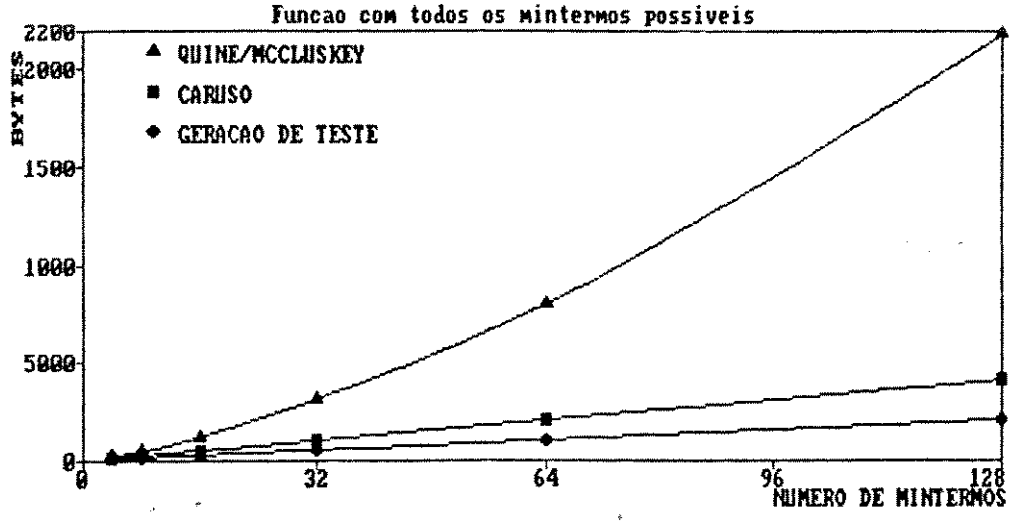


Figura III.36.- Comparação da quantidade de memória utilizada na simplificação das funções que contém todos os mintermos.

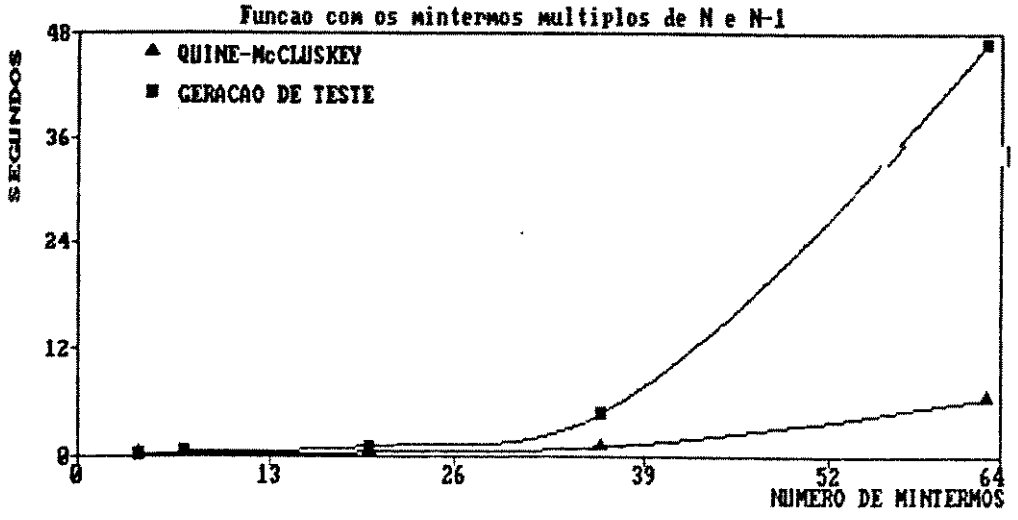


Figura III.37.- Comparação dos tempos de execução da simplificação das funções com os mintermos múltiplos de N e N-1.

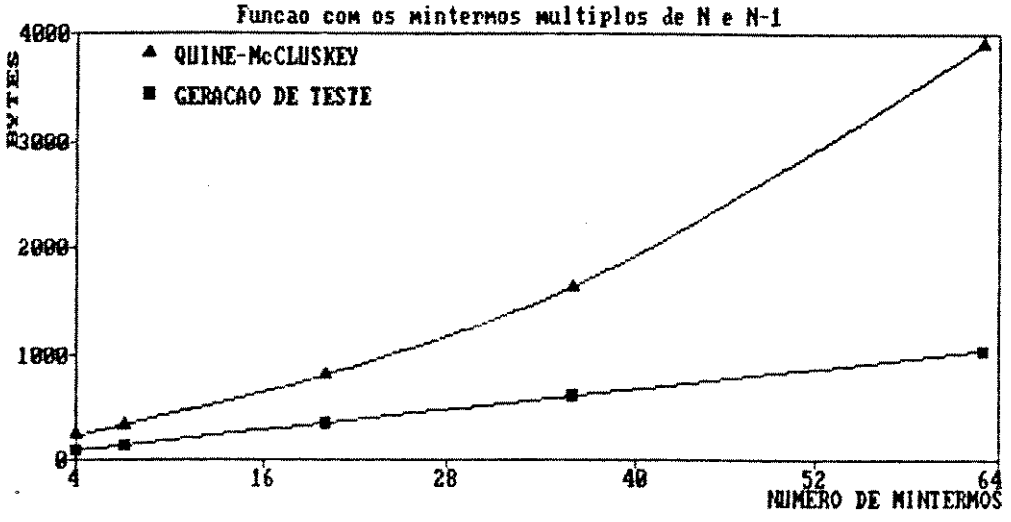


Figura III.38.- Comparação da quantidade de memória utilizada na simplificação das funções com os mintermos múltiplos de N e N-1

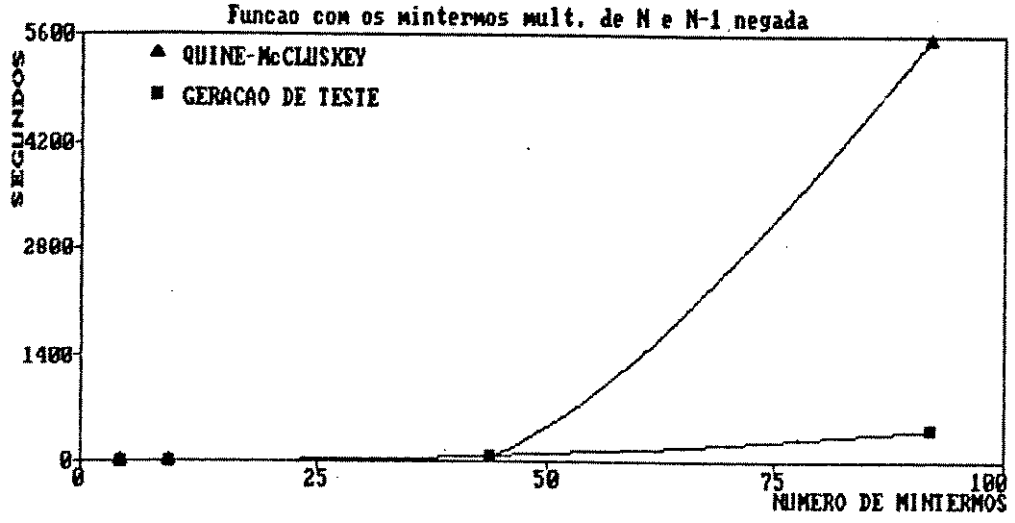


Figura III.39.- Comparação dos tempos de execução da simplificação das funções negadas daquelas contendo os mintermos múltiplos de N e N-1.

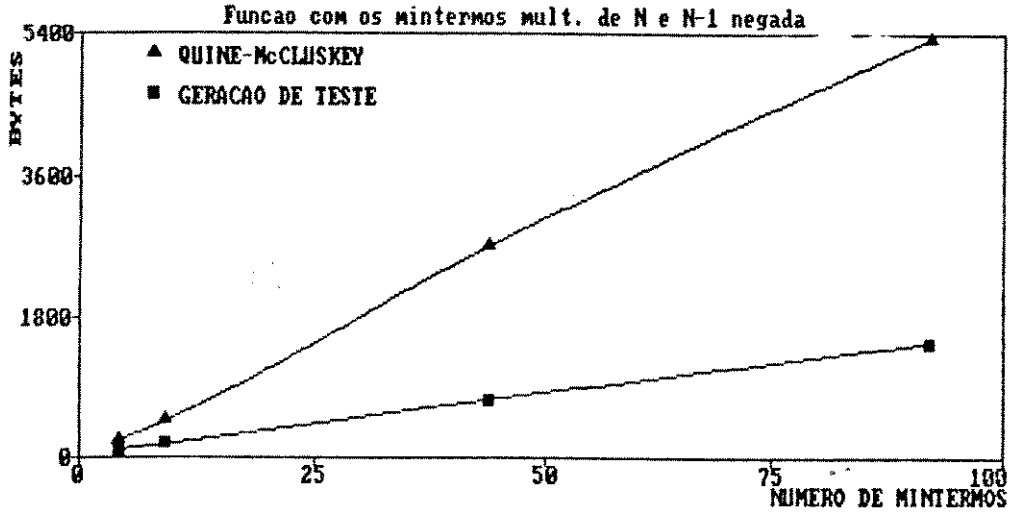


Figura III.40.- Comparação da quantidade de memória utilizada na simplificação das funções negadas daquelas contendo os mintermos múltiplos de N e N-1.

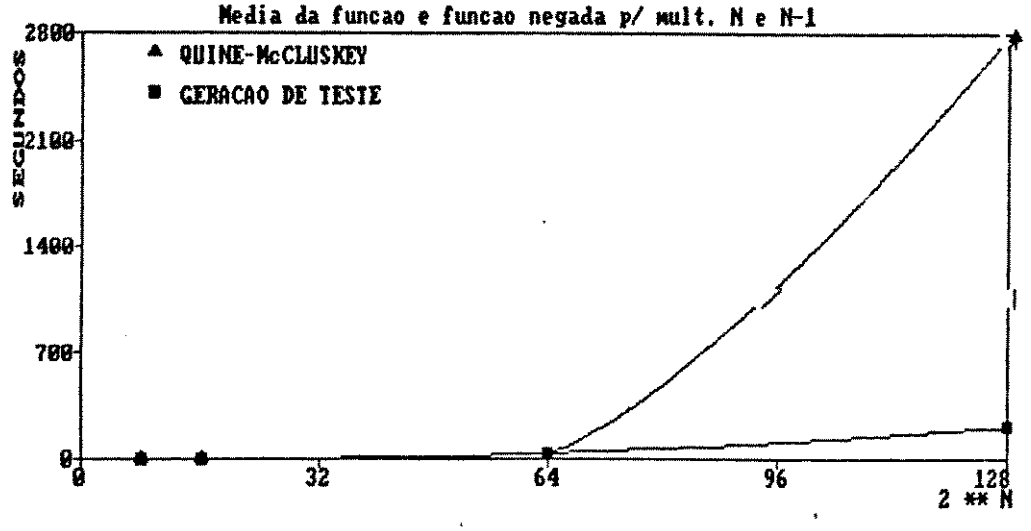


Figura III.41.- Comparação das médias dos tempos de execução da simplificação das funções que contém os mintermos múltiplos de N e N-1, e de suas funções negadas.

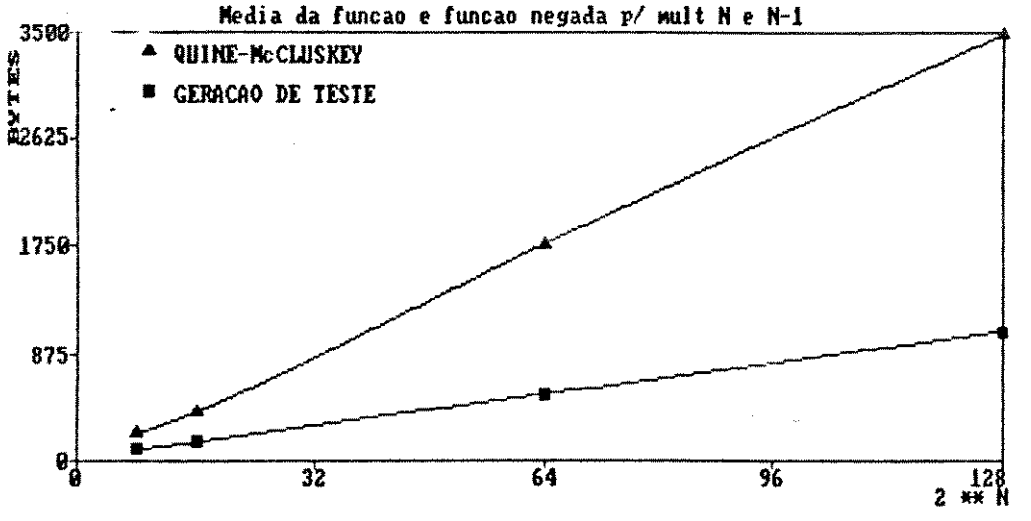


Figura III.42.- Comparação das médias da quantidade de memória utilizada na simplificação das funções com os mintermos múltiplos de N e N-1 e de suas funções negadas.

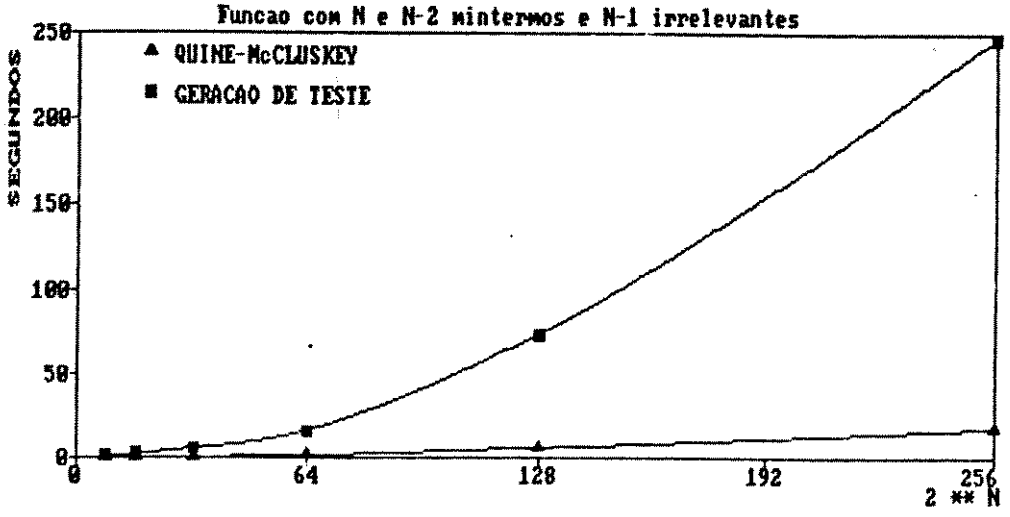


Figura III.43.- Comparação dos tempos de execução da simplificação das funções com mintermos múltiplos de N e N-2 e irrelevantes múltiplos de N-1.

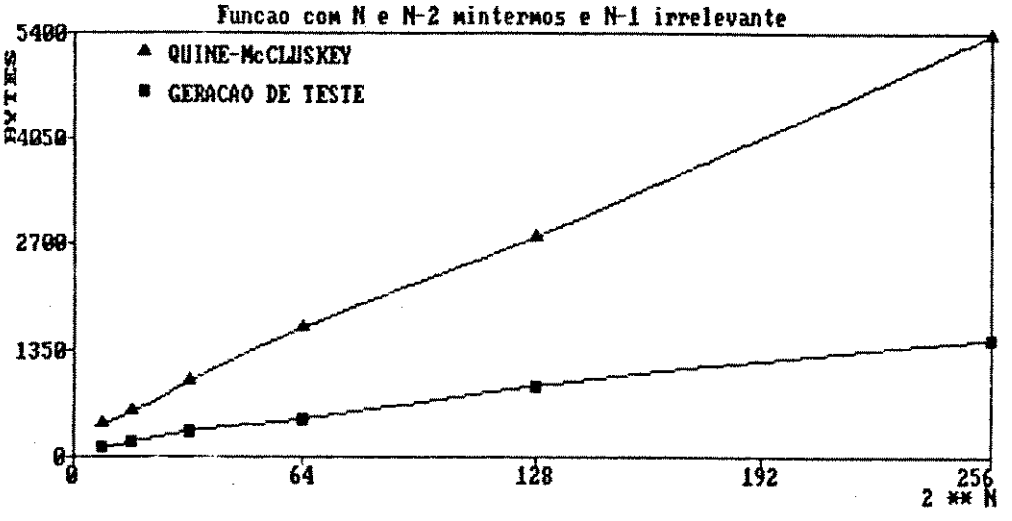


Figura III.44.- Comparação da quantidade de memória utilizada na simplificação das funções com mintermos múltiplos de N e N-2 e os irrelevantes múltiplos de N-1.

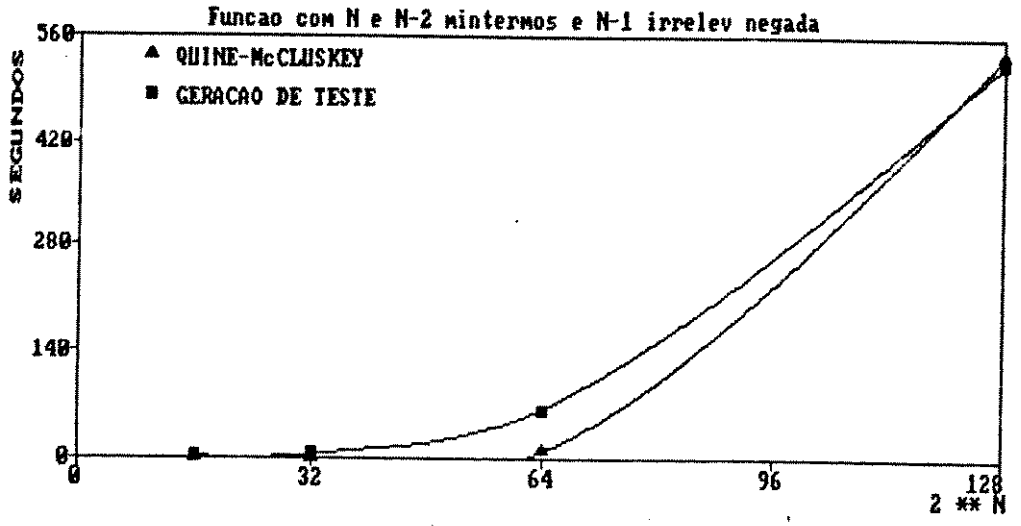


Figura III.45.- Comparação dos tempos de execução da simplificação das funções negadas daquelas cujos mintermos são múltiplos de N e N-2 e os irrelevantes são múltiplos de N-1.

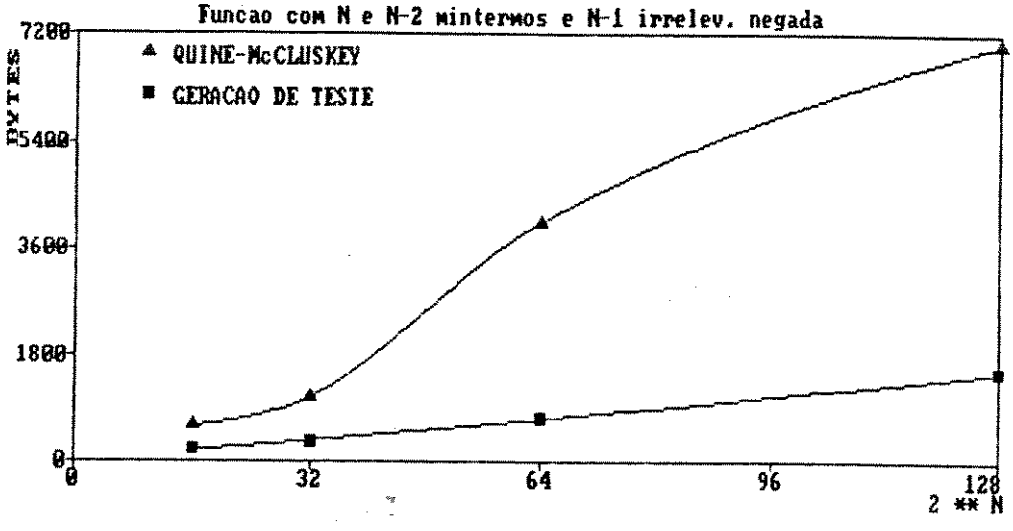


Figura III.46.- Comparação da quantidade de memória utilizada na simplificação das funções negadas daquelas cujos mintermos são múltiplos de N e N-2 e os irrelevantes são múltiplos de N-1.

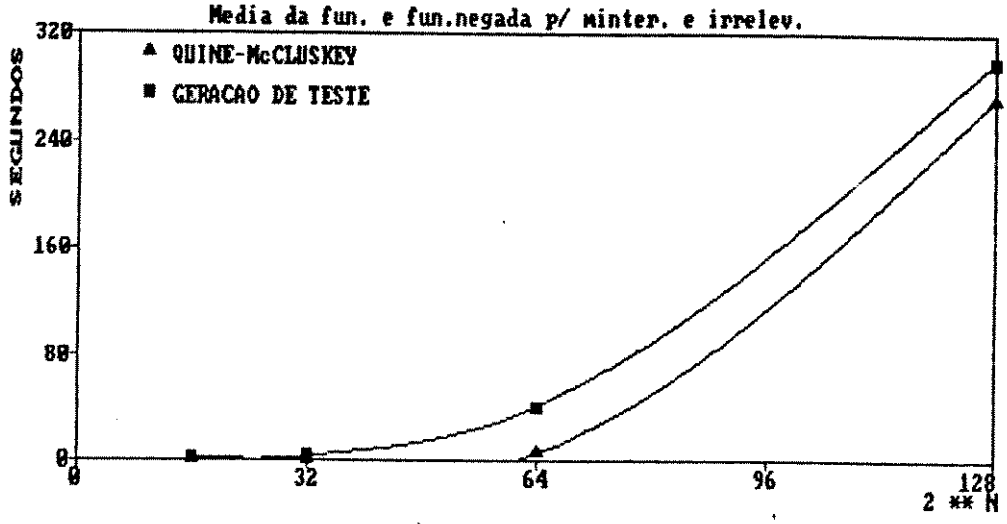


Figura III.47.- Comparação da média dos tempos de execução da simplificação das funções cujos mintermos são múltiplos de N e N-2 e os irrelevantes são os múltiplos de N-1 e de suas funções negadas.

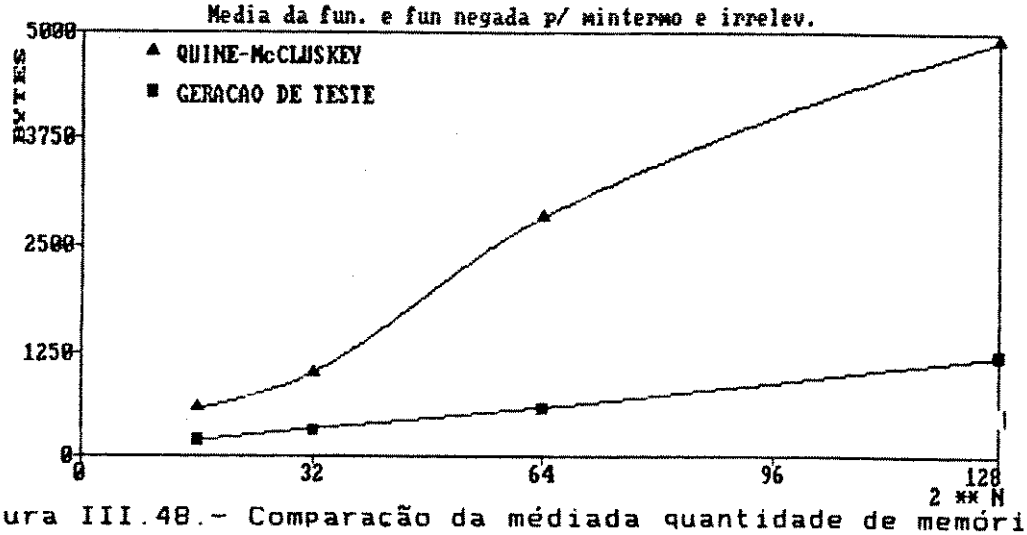


Figura III.48.- Comparação da médiada quantidade de memória utilizada na simplificação das funções cujos mintermos são múltiplos de N e N-2 e os irrelevantes são múltiplos de N-1 e de suas funções negadas.

O algoritmo "Geração" tem semelhanças com o de Quine-McCluskey e com o de Caruso. Da mesma forma que no algoritmo de Quine-McCluskey o algoritmo "Geração" obtem os implicantes primos e determina os implicantes primos essenciais. Entretanto, uma diferença marcante é que no algoritmo "Geração" a determinação dos implicantes primos (teste AND na versão simplificada) é acompanhada da cobertura dos mintermos envolvidos (teste OR na versão original). O teste OR final corresponde a eliminação de implicantes primos não essenciais resultantes da geração dos implicantes primos e correspondente cobertura.

Da mesma forma que no Caruso o algoritmo "Geração" utiliza uma ordenação inicial dos mintermos baseada no número de sucessores a distância 1, cujo objetivo é evitar testes em variáveis que seguramente são relevantes. Concomitante com a ordenação são identificados os mintermos isolados, isto é, aqueles que não possuem sucessores a distância 1, assim como são marcadas as variáveis para as quais existe vértices a distância 1. A operação teste AND no algoritmo "Geração" é bastante similar a operação INVBIT no algoritmo de Caruso.

O algoritmo de Quine-McCluskey determina uma solução de custo mínimo para a função, enquanto que os algoritmos de Caruso e Geração apenas garantem uma cobertura irredundante. Entretanto, na maior parte dos exemplos analisados todos os três métodos encontraram a solução de custo mínimo.

Dentre os 55 casos apresentados na seção III.8 apenas nos casos 4.10.0.2, 5.15.1.1, 5.24.0.1 e 6.31.0.1 o método Geração apresentou um custo maior que os métodos de Quine-McCluskey e Caruso.

III.9.- CONCLUSÃO

A minimização de funções booleanas com grande número de variáveis permanece um problema importante devido ao uso extensivo de PLAs (Estruturas Lógicas Programáveis) no projeto de circuitos digitais.

O enfoque clássico na minimização consiste na obtenção de todos os implicantes primos e na seleção dos implicantes primos essenciais que realizam a cobertura mínima da função.

Tais métodos não são úteis acima de uma dezena de variáveis, pois a geração dos implicantes primos exige grande tempo computacional e grande espaço de memória e a cobertura em geral consome mais tempo ainda.

Portanto é interessante o estudo de métodos de coberturas irredundantes com o objetivo de reduzir o uso de memória e tempo computacional.

O método "Geração" introduzido neste trabalho apresenta desempenho satisfatório, quando comparado com os métodos de Quine-McCluskey e de Caruso. A análise das figuras III.35 a III.48 mostra que o tempo de execução dos diferentes métodos é dependente do caso em questão, mas que frequentemente o método de Quine-McCluskey é o mais rápido entre os algoritmos analisados.

Quanto ao uso de memória a análise das figuras evidencia o melhor desempenho do método "Geração" para todos os casos analisados.

III.10.- REFERÊNCIAS BIBLIOGRÁFICAS

[1] E. J. McCluskey
Minimization of Switching Functions
Bell Syst.Tech.J.,vol.35,Nov.1956,pp.1417-1444.

[2] G. Caruso
A Local Selection Algorithm for Switching Function Minimization
IEEE Trans. on Computers, vol. C-33, No.1, Jan. 1984, pp.91-97

[3] A. C. R. Silva, M. C. G. Madureira, I. S. Bonatti
Cobertura Irredundante de Funções Booleanas Através da Geração de Padrão de Teste
IV SBmicro - Porto Alegre - Julho, 1989, pp. 427-439

[4] P. Goel
An Implicit Enumeration Algorithm to Generation Tests for Combinational Logic Circuits
IEEE Trans. Comput., vol C-30, Mar. 1981, pp. 215-222

[5] M.C.G.Madureira
Contribuição à Análise e Síntese de Circuitos Digitais
Tese de Mestrado-FEE-UNICAMP, Maio de 1987.

[6] M.Karnaugh
The Map Method for Synthesis of Combinational Logic Circuits
AIEE Trans.Commun.Electron.,vol.72,Nov.1953,pp.593-598.

[7] W.V.Quine
The Problem of Simplifying Truth Functions
Amer.Math.Mon.,vol.59,Oct.1952,pp.521-531.

[8] S.L. Hurst
Multiple-Valued Logic - Its Status and Its Future
IEEE Trans. on Computers, vol. C-33, No 12, Dec.1984.

C A P Í T U L O IV

MÁQUINAS SEQUENCIAIS SÍNCRONAS

IV.1.- INTRODUÇÃO [2], [3], [4]

O objetivo deste capítulo é apresentar um método sistemático para projeto de circuitos binários sequenciais síncronos.

Por circuito sequencial síncrono, entendem-se as máquinas sequenciais onde os elementos de memória são claramente identificáveis (Flip-Flops) e nos quais está presente uma entrada particular, denominada relógio, que sincroniza os eventos a que está sujeita esta máquina.

Assim, é perfeitamente identificável uma sequência enumerável de instantes através dos quais a descrição da máquina é realizada.

Na maioria dos sistemas digitais são necessários circuitos capazes de armazenar dados e de efetuar com eles algumas operações lógicas e ou numéricas. As saídas destes circuitos são funções das entradas primárias dos circuitos bem como da informação armazenada neles. Tais circuitos são chamados circuitos sequenciais.

Máquina sequencial, ou máquina de estados finitos, é o modelo abstrato do circuito sequencial real. Seu comportamento é descrito como uma sequência de eventos que ocorrem em instantes de tempo discretos designados por t1, t2, t3,... etc.

Suponha que uma máquina M receba alguns sinais de entrada e responda produzindo alguns sinais de saída. Se, no instante t, for aplicado um sinal x(t) a M, sua resposta z(t) depende de x(t) bem como das entradas de M nos instantes anteriores. Como a máquina M pode ter uma variedade infinita de passados possíveis, seria necessária uma capacidade de armazenamento infinita para guardá-los todos.

Este capítulo concentra-se nas máquinas cujo passado pode ser resumido em um conjunto finito de variáveis (Máquinas Determinísticas ou Markovianas).

Por exemplo, num somador binário serial, a resposta aos sinais de entrada

no instante t depende apenas destes sinais e do valor do transporte gerado no instante anterior. Assim, embora o somador possa ter um número infinito de passados, eles podem ser agrupados em duas grandes classes, aquelas que resultam em transporte igual a 1 e aquelas que resultam em transporte igual a 0 num dado instante.

Desta forma, são estudadas aqui as máquinas que podem distinguir entre um número finito de classes de entradas passadas, e estas classes são chamadas de estados internos destas máquinas. Assim, toda máquina de estados finitos contém um número finito de elementos de memória. Note que, embora restritos a máquinas com capacidade finita de armazenamento, não fica estabelecido nenhum limite para a duração da influência de uma entrada particular no comportamento futuro da máquina.

IV.2.- MÁQUINAS DE ESTADO

Uma máquina sequencial pode ser representada esquematicamente pelo circuito da figura IV.1. O circuito tem um número finito r de entradas. Os sinais aplicados a estas entradas constituem o conjunto das variáveis de entrada (X1,X2,...,Xr). Uma r-upla ordenada destes valores forma o vetor de entrada (ou simplesmente, entrada).

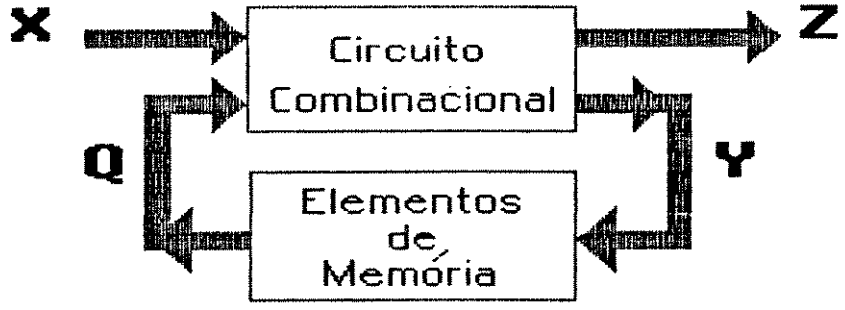


Figura IV.1.- Esquema de uma máquina sequencial.

Analogamente, o circuito tem um número finito m de saídas. Os sinais obtidos nestas saídas constituem o conjunto das variáveis de saída (Z1,Z2,...,Zm). Uma m-upla ordenada destes valores é o vetor de saída (ou saída).

O valor contido em cada elemento de memória é chamado de variável de estado, e o conjunto (Q1,Q2,...,Qk) constitue o conjunto das variáveis de estado. Os valores contidos nos k elementos de memória definem o estado

interno atual da máquina.

As saídas Z1,Z2,..Zm e as funções de transições internas Y1,Y2,..Ys são funções das entradas externas e do estado interno da máquina, e são definidas através do circuito combinacional mostrado na figura IV.1. Os valores Y, que aparecem nas saídas do circuito combinacional no instante t, determinam os valores das variáveis de estado no instante t+1, e portanto definem o próximo estado da máquina.

IV.2.1.- TABELAS E DIAGRAMAS DE ESTADO

As relações entre entradas, estados presentes ou atuais, saídas e próximos estados podem ser descritas por um diagrama de estados, ou por uma tabela de estados ou ainda, por uma tabela de transição.

Uma tabela de estados tem $p = 2^r$ colunas, uma para cada ocorrência do vetor de entrada, e $n = 2^k$ linhas, uma para cada estado.

Cada combinação de entradas e estados presentes determinam a saída produzida e o próximo estado para a máquina .

O diagrama de estados é um grafo orientado, onde cada estado da máquina corresponde a um nó. De cada nó emanam p arcos orientados, correspondendo às transições de estados causadas pela ocorrência da entrada. Cada arco orientado é rotulado com a entrada que determina aquela transição e com a saída gerada.

As máquinas que determinam o próximo estado Q(t+1) unicamente com base no estado Q(t) e na entrada X(t) atuais são chamadas de máquinas determinísticas ou máquinas markovianas.

Nas máquinas determinísticas,

$$Q(t+1) = f[\ Q(t),X(t)\ ] \qquad (\ IV.1\ )$$

onde f é a função de transição de estados. O valor da saída Z(t) é função do estado presente Q(t) e, muitas vezes, das entradas presentes X(t), ou seja:

$$Z(t) = g[\ Q(t)\ ] \qquad (\ IV.2\ )$$

ou,

$$Z(t) = g[\ Q(t),X(t)\ ] \qquad (\ IV.3\ )$$

onde g é a função saída.

Uma máquina com as propriedades descritas nas equações (IV.1) e (IV.2) é conhecida como máquina de Moore e uma máquina com as características das equações (IV.1) e (IV.3) é conhecida como máquina de Mealy.

IV.2.2.- ELEMENTOS DE MEMÓRIA E TABELAS DE TRANSIÇÃO

Através do uso de elementos de memória, denominados multivibradores biestáveis ou flip-flops, projeta-se um circuito que opera de acordo com as especificações de uma dada tabela. Os estados destes elementos são associados aos estados da máquina por um processo chamado alocação de estados.

Uma tabela de transição especifica o próximo estado dos elementos de memória para cada combinação de entradas e variáveis de estado.

Para gerar os próximos estados da maneira especificada, os elementos de memória devem ser supridos com as entradas apropriadas. A função lógica que descreve o efeito das entradas de um elemento de memória e de seu estado presente na determinação de seu próximo estado é chamada de função característica do elemento de memória (H), definida por :

$$Q(t+1) = H[\, Q(t), Y(t) \,] \quad .$$

Por exemplo, para o flip-flop tipo D, conforme a tabela IV.1 :

$$Q(t+1) = Y(t) = D(t)$$

Para o flip-flop JK, conforme a tabela IV.2 :

$$Q(t+1) = Q'(t).J(t) + Q(t).K'(t)$$

Assim, para efeito de projeto das máquinas o problema se resume à escolha dos elementos de memória e à síntese de suas entradas Y(t).

Q(t)	Q(t+1)	D(t)
0	0	0
0	1	1
1	0	0
1	1	1

Tabela IV .1.- Função característica do flip-flop D.

Q(t)	Q(t+1)	J(t)	K(t)
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Tabela IV.2.- Função característica do flip-flop JK.

Note na tabela IV.2 que, para algumas transições, há entradas irrelevantes (X) que se transformam em don't care states na síntese da função lógica correspondente. Isto proporciona, normalmente, circuitos combinacionais mais simples quando utilizam-se flip-flops JK ao invés de D.

As máquinas sequenciais onde aparecem estados irrelevantes são chamadas de máquinas sequenciais incompletamente especificadas.

IV.3.- SISTEMATIZAÇÃO DE PROJETO

O modelamento de um circuito sequencial pode ser feito por um diagrama de estado. Tal diagrama define as transições de estado e suas saídas, quando submetidas às entradas.

Cada estado na tabela é representado por um código binário único (alocação de estados) e a "tabela de estados alocados" resultante é analisada com o objetivo de definir as " funções de transições internas " e as " funções de saída". A partir destas funções é possível a realização física do circuito e as funções podem ser implementadas usando elementos lógicos padrões, como gates.

O problema da alocação de estados é bastante relevante no que diz respeito ao desempenho e custos dos circuitos finais.

O número de alocações possíveis é bastante grande para permitir uma enumeração completa de todas elas como um método viável de solução. Um grande número de procedimentos para a obtenção de alocações iniciais de estados "boas" ou "quase ótimas", têm sido propostos nesses últimos anos.

A sistematização e automação do projeto de circuitos digitais é extremamente conveniente pois retira do projetista as tarefas cansativas, sujeitas a erros e demoradas, para permitir-lhe uma análise crítica das possíveis soluções. é neste contexto que é apresentado o programa TABELA.

IV.3.1.- PROGRAMA TABELA

O programa TABELA, inicialmente apresentado por Madureira [1], foi melhorado nos aspectos de interação com o usuário e no desempenho do algoritmo de minimização das funções booleanas.

O programa TABELA (figura IV.2) gera a tabela de transição de uma máquina sequencial a partir de seu diagrama de estados e minimiza as funções de transições internas correspondentes aos elementos de memória utilizados e as funções de saída do circuito.

Os dados requisitados pelo programa são:

Nome do dispositivo de saída de resultados;
Número de flip-flops;
Tipo de cada um deles (D ou JK);
Número de variáveis de entrada;
Número de variáveis de saída;
Tabela de estado, na forma:
estado atual, próximo estado, entrada, saída.

Os estados, as entradas e as saídas devem estar na forma decimal. As máquinas podem ser completa ou incompletamente especificadas, e devem estar na representação de Mealy.

O programa monta a tabela de transição armazenando-a no arquivo de saída. A partir desta tabela são obtidos os mintermos e os don't care states das funções de transições internas de todos os flip-flops e da saída do circuito. Utilizando-se um algoritmo de minimização de funções booleanas, estas funções são obtidas nas suas fórmulas mínimas.

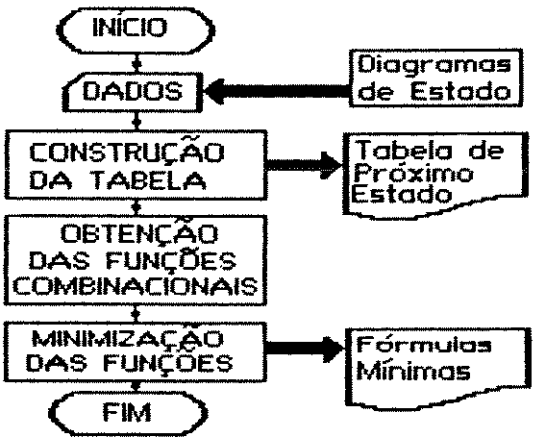


Figura IV.2.- Diagrama de blocos do programa Tabela.

IV.4 - EXEMPLO DE UM PEQUENO PROJETO

Para ilustrar as etapas do projeto de um circuito lógico, estuda-se como exemplo um detector de sequências de 3 zeros consecutivos sem sobreposição, isto é, o circuito deve produzir uma saída Z = 1, quando a sequência de entrada contiver 3 zeros consecutivos, conforme ilustrado na sequência

seguinte:

X = 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1

Z = 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0

Figura IV.3.- Diagrama de blocos do detector da sequência de 3 zeros consecutivos.

a) Projeto intuitivo

A figura IV.4 apresenta um circuito de um detector de 3 zeros consecutivos, que armazena a sequência de chegada em um registro de deslocamento de 3 bits e analisa as saídas deste registro. Quando os 3 flip-flops contiverem zero em suas saídas, a saída Z é feita igual a um. Para que a detecção seja sem sobreposição um pulso de "set" é aplicado nos flip-flops cada vez que Z = 1.

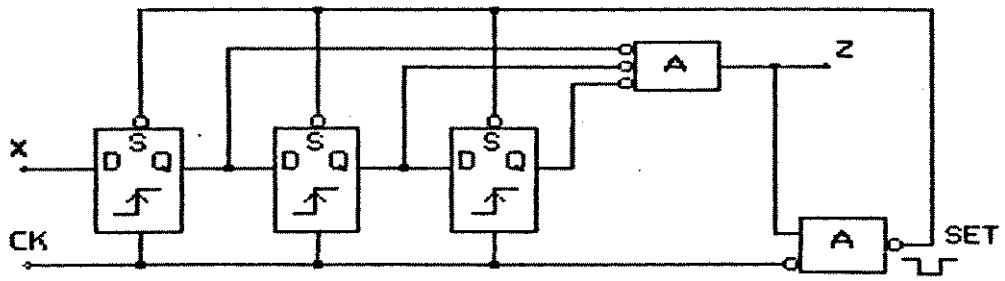


Figura IV.4.- Circuito intuitivo de um detector de 3 zeros consecutivos.

b) Detector de Mealy

A figura IV.5 apresenta o diagrama de estado da máquina de Mealy que detecta 3 zeros consecutivos sem sobreposição. O estado A é considerado o estado inicial, e os estados A,B e C formam o ciclo que detecta a sequência de 3 zeros consecutivos. Note que quando o detector está no estado B e recebe um 1 na entrada retorna para o estado A, o mesmo acontecendo para o estado C.

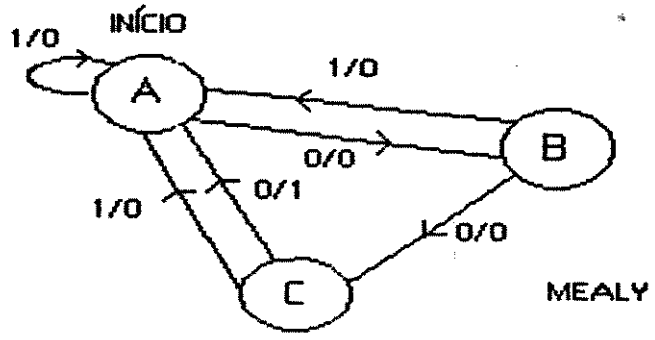


Figura IV.5.- Máquina de Mealy de um detector de 3 zeros consecutivos.

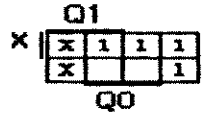
A figura IV.6 apresenta a tabela de próximo estado do detector de Mealy, e a tabela resultante da seguinte alocação de estados:

Estado	Q1	Q0
A	1	1
B	0	1
C	0	0

A figura IV.6 mostra ainda os mapas de Karnaugh, e suas correspondentes expressões lógicas. A escolha da alocação é arbitrária e qualquer outra pode ser utilizada. é importante ressaltar que diferentes alocações podem dar origem a diferentes circuitos, isto é, circuitos com uma parte combinacional contendo um número maior ou menor de portas lógicas.

X	ESTADO⁻	ESTADO⁺	Z
0	A	B	0
0	B	C	0
0	C	A	1
1	A	A	0
1	B	A	0
1	C	A	0

X	Q1⁻	Q0⁻	Q1⁺	Q0⁺	Z
0	1	1	0	1	0
0	0	1	0	0	0
0	0	0	1	1	1
1	1	1	1	1	0
1	0	1	1	1	0
1	0	0	1	1	0



$D1 = X + \bar{Q}0$

$D0 = X + \bar{Q}0 + Q1$

$Z = \bar{X} \cdot \bar{Q}0$

Figura IV.6.- Tabelas de estados e mapas de Karnaugh do detector de Mealy.

A figura IV.7 mostra o detector de Mealy realizado com flip-flops D e a figura IV.8 mostra o diagrama de estado feito a partir do circuito da figura IV.7 com o intuito de conferir o projeto. Note que na figura IV.8 aparece um quarto estado 10 que não existia na figura IV.5. é importante que este estado esteja conexo aos demais estados, pois caso contrário a realização física da máquina poderia conter ciclos secundários que, se atingidos, por exemplo ao ligar-se o circuito, não poderiam mais ser abandonados, inviabilizando o uso desta máquina.

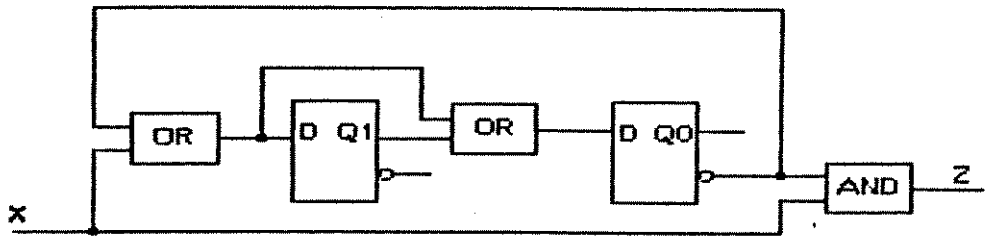


Figura IV.7.-Circuito de Mealy de um detector de 3 zeros consecutivos.

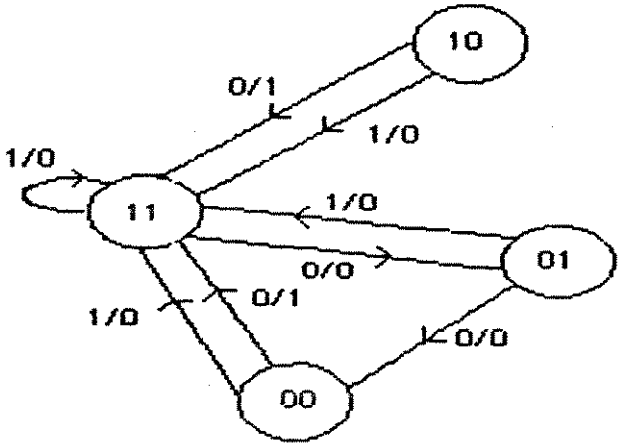


Figura IV.8.- Máquina de estado feita a partir do circuito da figura IV.7.

c) Detector de Moore

A figura IV.9 mostra a máquina de Moore para o detector de 3 zeros consecutivos. O estado A foi considerado o estado inicial. Note que o ciclo principal da máquina é composto pelos estados B, C e D. Note ainda que por ser uma máquina de Moore a saída depende apenas do estado.

é interessante notar que o detector de Moore contém 4 estados, enquanto o

de Mealy apenas 3. A máquina de Moore contém, em geral, mais estados que a correspondente máquina de Mealy.

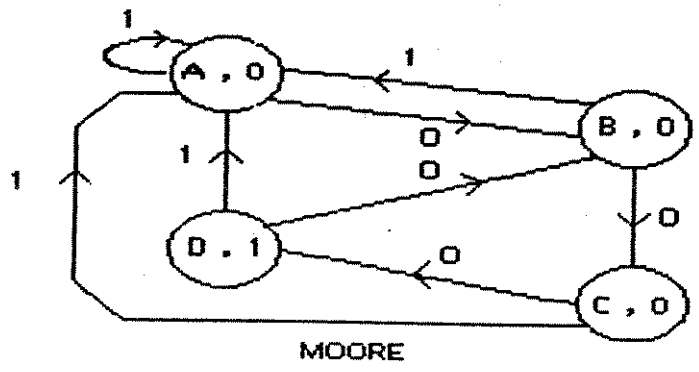


Figura IV.9.- Máquina de Moore de um detector de 3 zeros consecutivos.

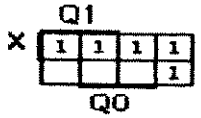
A figura IV.10 apresenta as tabelas de próximo estado do detector de Moore, onde a alocação utilizada é mostrada a seguir:

Estado	Q1	Q0
A	1	1
B	0	1
C	0	0
D	1	0

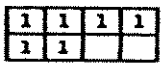
A figura IV.10 mostra ainda os mapas de Karnaugh e as correspondentes expressões lógicas da síntese da máquina de Moore feitas utilizando-se flip-flops D.

X	ESTADO$^-$	ESTADO$^+$	Z
0	A	B	0
0	B	C	0
0	C	D	0
0	D	B	1
1	A	A	0
1	B	A	0
1	C	A	0
1	D	A	1

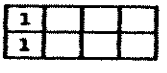
X	Q1$^-$	Q0$^-$	Q1$^+$	Q0$^+$	Z
0	1	1	0	1	0
0	0	1	0	0	0
0	0	0	1	0	0
0	1	0	0	1	1
1	1	1	1	1	0
1	0	1	1	1	0
1	0	0	1	1	0
1	1	0	1	1	1



$D1 = X + \bar{Q}1 \cdot \bar{Q}0$



$D0 = X + Q1$



$Z = Q1 \cdot \bar{Q}0$

Figura IV.10.- Tabelas de estados e mapas de Karnaugh para máquina de Moore.

A figura IV.11 mostra o circuito que realiza o detector de Moore. A partir deste circuito pode-se realizar o diagrama de estados para validação do projeto.

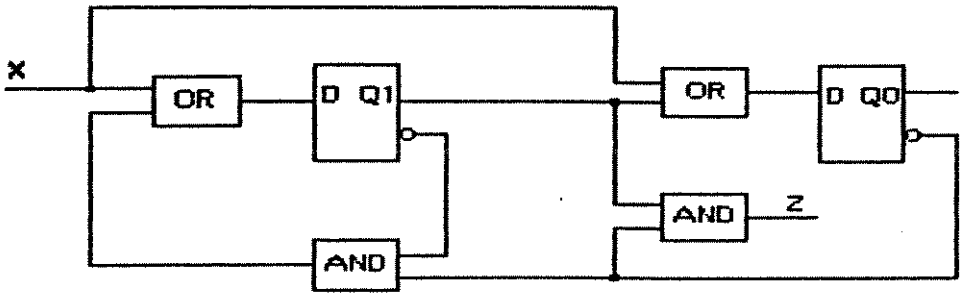


Figura IV.11.- Circuito que realiza o detector de Moore.

É interessante observar que os circuitos das figuras IV.4 (intuitivo), IV.7 (Mealy) e IV.11 (Moore) têm complexidades similares.

IV.5 - REDUÇÃO DE ESTADOS

A tradução da descrição verbal de um sistema para a sua correspondente máquina de estado resulta, em geral, em uma máquina com alto grau de redundância. Quase sempre é possível realizar o sistema com uma máquina equivalente com menos estados. Por máquina equivalente entende-se uma máquina que produza as mesmas saídas para as correspondentes mesmas entradas.

Exemplo 1.- Detector da sequência 10010 com sobreposição:

A sequência a seguir ilustra um exemplo da atuação do detector.

X = 1 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1
Z = 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0

A figura IV.12 apresenta uma máquina de estado de Mealy do detector da sequência 10010 com sobreposição. Note que os estados C,D e E formam o ciclo principal desta máquina. O estado A foi considerado o estado inicial da máquina. Note que esta máquina pode apresentar um efeito transitório, isto é, se o estado inicial for diferente de A, a máquina pode produzir uma saída ativa equivocada, mas isto ocorrerá apenas uma vez.

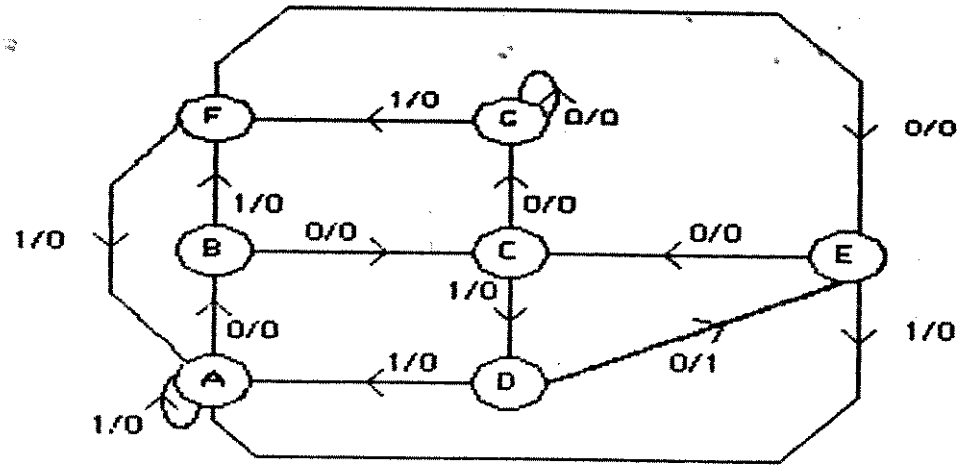


Figura IV.12.- Máquina de Mealy para o detector da sequência 10010.

A tabela IV.3 apresenta a tabela de próximo estado da máquina da figura IV.12.

X	Q^-	Q^+	Z
0	A	B	0
0	B	C	0
0	C	G	0
0	D	E	1
0	E	C	0
0	F	E	0
0	G	G	0
1	A	A	0
1	B	F	0
1	C	D	0
1	D	A	0
1	E	A	0
1	F	A	0
1	G	F	0

Tabela IV.3.- Tabela de próximo estado do detector da sequência 10010.

A aplicação de um algoritmo de partição na máquina da figura IV.12 permite verificar se esta possui estados redundantes.

Partição de equivalência é a definição de um conjunto de classes de

equivalência, tal que estados em uma mesma classe são equivalentes e estados em classes distintas são distinguíveis.

Dois estados são equivalentes se, para as mesmas situações da entrada, tiverem os próximos estados dentro da mesma classe de equivalência e produzirem as mesmas saídas.

Os estados A e F, na figura IV.12, são equivalentes e os estados C e D são distinguíveis. Para validar esta afirmação aplica-se um algoritmo derivado da definição de partição de equivalência.

ALGORITMO DE PARTIÇÃO DE EQUIVALÊNCIA

a) Partição do tipo 1

Cada classe de estados produz as mesmas saídas para as mesmas condições das entradas.

b) Partição do tipo 2

Para uma dada entrada os próximos estados de uma classe estão em uma mesma classe de estados.

A obtenção da máquina mínima, isto é, com o menor número de classes de estados, é realizada supondo-se inicialmente que todos os estados da máquina original sejam equivalentes entre si. A aplicação da partição do tipo 1 na máquina com uma única classe de estados resulta numa máquina com tantas classes de estados quantas forem as situações distintas da saída, para uma dada situação da entrada.

No exemplo da figura IV.12, tem-se:

Máquina inicial: (ABCDEFG)

Aplicando-se a partição do tipo 1, tem-se: (ABCEFG , D)

Note, na tabela IV.3, que para o estado D a saída Z é distinta das saídas associadas aos demais estados.

A figura IV.13 apresenta um diagrama de estado supondo-se que os estados A,B,C,E,F e G formam uma classe de equivalência 1 e que o estado D forma uma classe 2.

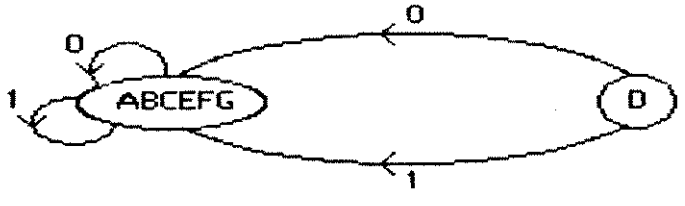


Figura IV.13.- Máquina de estado com 2 classes de estados.

A máquina de estado da figura IV.13 não é equivalente à máquina de estado

original, figura IV.12. Isto porque, na máquina original, o estado C para entrada X = 1 tem como próximo estado o estado D e na máquina da figura IV.13 o próximo estado de C está na classe que não contém o estado D. Portanto a classe que contém os estado A,B,C,E,F e G não é uma classe de equivalência pois o estado C é distinguível dos demais desta classe.

A aplicação da partição do tipo 2 consiste na verificação da existência de estados distintos dentro de uma suposta classe de equivalência.

Aplicando-se a partição do tipo 2 na máquina da figura IV.13, obtém-se a máquina da figura IV.14 composta de três classes: (ABEFG , D , C).

A máquina de estados da figura IV.14 descreve a partição atual, que supõe que as 3 classes resultantes são classes de equivalências. é necessário verificar se esta partição em 3 classes corresponde à máquina original, isto é, se todos os estados de uma certa classe são efetivamente estados equivalentes.

Isto não ocorre, pois na classe ABEFG há estados distinguíveis entre si. Em particular, os estados BE são distinguíveis dos estados AFG, pois quando X = 0, os próximos estados da classe BE estão na classe C e os próximos estados da classe AFG estão na classe ABEFG.
Assim a aplicação da partição do tipo 2 resulta em: (AFG , D , C , BE)

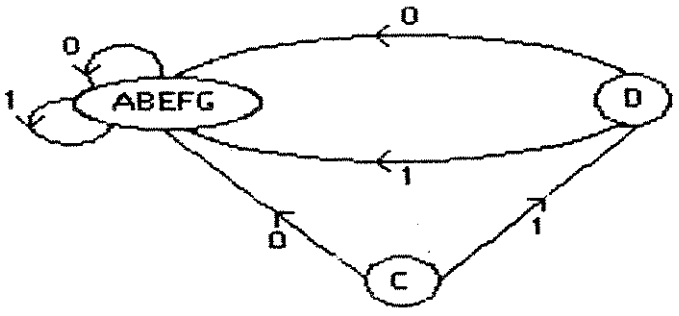


Figura IV.14.- Máquina de estado com 3 classes de estados.

Aplicando-se novamente a partição do tipo 2, na máquina resultante verifica-se que o estado G não é equivalente aos estados AF. Assim a partição do tipo 2 resulta em : (AF , D , C , BE , G).

A figura IV.15 apresenta a máquina reduzida ou mínima, isto é, nesta máquina os estados A e F são equivalentes entre si, pois o próximo estado, quando X = 0, da classe AF é a classe BE e quando X = 1, é a própria classe AF, e além disso a saída Z é igual a zero, para as duas situações da entrada, ou seja, não é possível distinguir os estados A e F através da observação da

saída da máquina. Da mesma forma os estados B e E são equivalentes entre si.

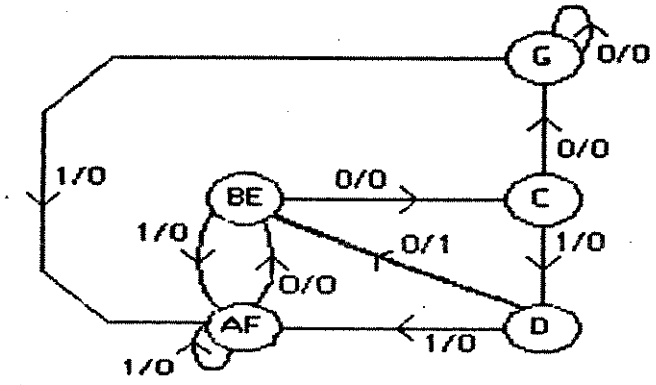


Figura IV.15.- Máquina de estado mínima equivalente à máquina original da figura IV.12.

A aplicação do algoritmo de partição na máquina de estados da figura IV.16 ilustra mais um exemplo de redução de estados.

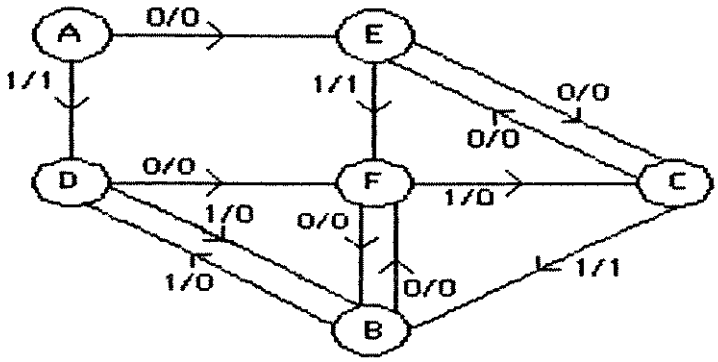


Figura IV.16.- Máquina de estado do exemplo 2.

A figura IV.17 apresenta a tabela de estados da máquina da figura IV.16. Apresenta ainda a sequência de máquinas resultantes da aplicação das partições do tipo 1 e do tipo 2.

X	Q^-	Q^+	Z
0	A	E	0
0	B	F	0
0	C	E	0
0	D	F	0
0	E	C	0
0	F	B	0
1	A	D	1
1	B	D	0
1	C	B	1
1	D	B	0
1	E	F	1
1	F	C	0

PARTIÇÃO DO TIPO 1
(BDF , ACE)
PARTIÇÃO DO TIPO 2
(BD . ACE . F)
PARTIÇÃO DO TIPO 2
(BD . AC . F . E)

Figura IV.17.- Tabela do próximo estado e aplicação da partição na máquina da figura IV.16.

A figura IV.18 mostra a máquina de estados reduzida equivalente à máquina da figura IV.16, obtida pela aglutinação dos estados A e C e dos estados B e D.

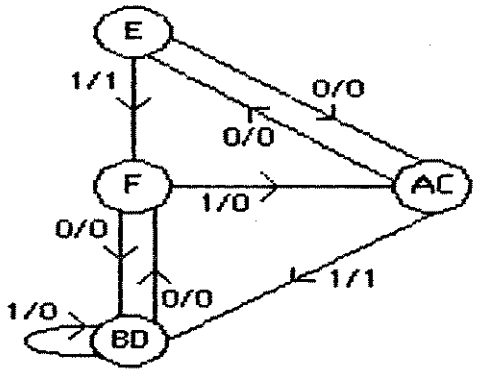


Figura IV.18.- Máquina de estado mínima equivalente à da figura IV.16.

IV.6.- CONCLUSÃO

Neste capítulo foi reapresentado o programa TABELA que realiza o projeto sistemático de máquinas sequenciais síncronas. O programa TABELA foi melhorado nos seus aspectos conversacionais.

Este programa retira do projetista as tarefas cansativas e repetitivas de testar e realizar diferentes alocações de estado no projeto de uma dada máquina, permitindo concentrar-se na análise crítica das diferentes soluções.

IV.7.- REFERÊNCIAS BIBLIOGRÁFICAS

[1] M. C. G. Madureira
Contribuição à Análise e Síntese de Circuitos Digitais
Tese de Mestrado - FEE-UNICAMP, Maio de 1987

[2] Z. Kohavi
Switching and Finite Automata Theory
McGraw Hill, 1970

[3] F. Hill, G. Peterson
Introduction to Switching Theory & Logical Design
John Wiley & Sons, 1974

[4] I.S. Bonatti, M.C.G. Madureira
Introdução à Análise e Síntese de Circuitos Lógicos
Editôra Unicamp, no prelo.

CAPÍTULO V

SIMULAÇÃO LÓGICA

V.1.- INTRODUÇÃO [3], [4], [5], [6]

No contexto de projeto de circuitos digitais, observa-se uma tendência cada vez maior no sentido de se projetar circuitos na forma de circuitos integrados de alta escala e de acordo com as especificações do usuário, o que leva a circuitos cada vez mais complexos.

A simulação de circuitos digitais representa, cada vez mais, uma diminuição no custo e no tempo de projeto, pois permite analisar o comportamento do circuito, verificando seu funcionamento e a possível existência de erros de projeto, antes de sua implementação. Isto torna mais simples e baratas as modificações necessárias no projeto.

Os primeiro simuladores lógicos foram desenvolvido na década de 60 e utilizavam um modelo sem associação de atrasos de propagação nas portas lógicas. A partir da década de 70 foram desenvolvidos simuladores considerando atrasos de propagação numa crescente sofisticação de modelos de tempo [1].

Um grande número de empresas, especialmente norte-americanas,oferece hoje no mercado simuladores lógicos cujos preços variam de 500 a dezenas de milhares de dólares.

A simulação é um método tradicional de validação de projetos. Ela exige a determinação de um conjunto de vetores de excitação e resultados esperados significativos, cobrindo em princípio todas as condições reais possíveis, o que é extremamente difícil de ser determinado.

Na secção V.2 é apresentado um programa que simula circuitos digitais ao nível de portas lógicas. A idéia utilizada foi a de se criar um programa com o qual o usuário descreve, através de sintaxe apropriada, o circuito e suas entradas (excitações) e obtém as respostas do circuito nos pontos de seu interesse.

Estas respostas podem ser obtidas de duas formas. Na forma numérica, onde é gerado um arquivo com a descrição de todas as transições do circuito, os instantes em que ocorrem, e os valores "0" e "1" para os respectivos níveis lógicos; ou na forma gráfica, onde são visualizadas as formas de onda das entradas e saídas do circuito.

O usuário pode selecionar as ondas que deseja e delimitar os intervalos de tempo em que serão mostradas as ondas escolhidas, podendo ampliar ou reduzir janelas de tempo, e outras facilidades.

A idéia básica do programa é descrever os circuitos mais simples, como as portas lógicas isoladas, e utilizá-los como elementos básicos, evoluindo-se para circuitos mais complexos, como flip-flops, shift-registers, etc. Os quais, por sua vez, são guardados em uma biblioteca para uso em outros circuitos mais complexos.

V.2.- O SIMULADOR LÓGICO [2], [6]

V.2.1.- ESTRUTURA GERAL

O simulador LÓGICO foi inicialmente desenvolvido por Madureira [2], e foi melhorado nos seus aspectos de interação com o usuário, na sua biblioteca de funções e na manipulações das filas de eventos discretos.

O simulador constitui-se num processo no qual são definidos estados, transições entre estados, entradas e saídas. As entradas do processo são as entradas primárias do circuito, os estados são as variáveis de saída de cada uma de suas células básicas, e as saídas são as saídas primárias do circuito. As transições entre os estados do processo ocorrem devido às transições nas entradas. A regra de transição entre os estados é resultante da estrutura do circuito, de seu estado anterior e dos valores das entradas anteriores à transição.

O LÓGICO é um simulador estrutural e funcional. Estrutural porque a simulação é baseada em rotinas que simulam os gates básicos de circuitos tanto combinacionais como sequenciais com os quais se pode construir circuitos mais complexos, formando uma estrutura piramidal de uso. Estas rotinas, por sua vez, realizam todo o tratamento funcional dos elementos básicos, constituindo-se em "primitivas", ou seja, na elaboração de circuitos mais complexos não é necessário levar-se em conta suas propriedades funcionais, mas apenas suas estruturas.

Todas as portas lógicas apresentam um certo tempo de resposta, isto é, um

tempo de propagação da informação da entrada para a saída. O simulador LÓGICO permite que se defina o tempo de atraso de cada gate do circuito. São definidos dois atrasos para cada gate, um na transição de nível baixo para alto (TPLH) e um de alto para baixo (TPHL). Estes valores podem ser definidos individualmente para cada gate, ou simultaneamente para todos os gates, ou ainda podem ser estabelecidos limites inferiores e superiores dentro dos quais o programa atribuirá valores aleatórios de atraso.

Para o tratamento algorítmico dos atrasos, foi criado o conceito de evento. Um evento é uma transição que ocorre numa entrada ou numa saída qualquer do circuito. Os instantes inicial e final da simulação constituem-se em eventos especiais.

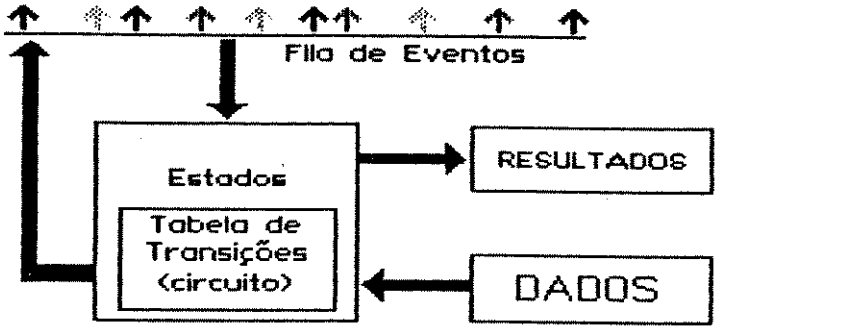


Figura V.1.- Processo de Simulação

A simulação resume-se na resolução e na geração de eventos que deverão estar ordenados numa fila de acordo com o instante em que ocorrem. O funcionamento do simulador é ilustrado na figura V.1 e processa-se da forma explicada a seguir.

Um vetor V de variáveis do tipo "byte" (0..255) armazena os valores das entradas e os estados de todos os gates do circuito em um dado instante. Cada gate é referenciado sempre pelo índice que o representa neste vetor. Convencionou-se que índices negativos do vetor representam entradas primárias do circuito e índices positivos seus estados internos.

Quando um evento é uma transição na entrada, os valores desta são atualizados e o circuito é "executado", ou seja, a partir dos valores das entradas são obtidos os valores de saida do circuito através da chamada das rotinas correspondentes.

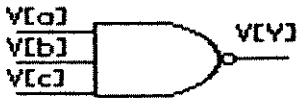
Para todos os eventos do tipo transição interna, que ocorrerem num mesmo

instante, os valores das posições do vetor correspondentes a estes pontos do circuito são atualizados e então o circuito é executado.

As rotinas básicas definidas não atualizam diretamente as posições do vetor de saída mas geram eventos, em instantes determinados pelo tempo de atraso das portas, que serão inseridos na fila na posição correta e tratados no devido tempo. Assim que um evento é tratado, passa-se ao próximo da fila. Isto é mostrado na figura V.2 onde são descritos um gate NAND de 3 entradas e um gate OR de 2 entradas, como exemplos.

As rotinas que simulam os gates calculam o nível lógico da saída de acordo com a função lógica simulada, e dos valores lógicos de suas entradas.

```
PROCEDURE NAND3(a,b,c,Y);
BEGIN
 E:= V[a]*V[b]*V[c];
 IF E>=2 THEN E:=2 ELSE E:=1-E;
 (*COLOCA V[Y]:=E NA FILA *)
END;
```




```
PROCEDURE OR2(a,b,Y);
BEGIN
 E:= V[a] OR V[b];
 IF E=3 THEN E:=1;
 (*COLOCA V[Y]:=E NA FILA *)
END;
```

Figura V.2.- Rotinas Básicas

O algoritmo trabalha com valores aritméticos 0, 1 e 2, representando os níveis lógicos '0', '1' e 'X' respectivamente. O estado 'X' (indeterminado) representa o não conhecimento do nível lógico da porta. A lógica de 3 valores associada é mostrada na figura V.3. Tal lógica pode ser expandida para quatro (ou mais) valores se se desejasse, por exemplo, tratar de circuitos "tristate", incluindo-se um estado de alta impedância.

V.2.2.- CIRCUITOS COMBINACIONAIS

A implementação dos gates lógicos define a convenção de chamada das rotinas pelo usuário. O exemplo seguinte ilustra um caso de um gate NAND de n entradas :

NANDn(V1,V2,..,Vn,Y); onde, n é o número de entradas da porta;

V1,V2,..,Vn são os índices das n entradas no vetor V; Y é o índice da saída no vetor V .

0 = Falso
1 = Verdadeiro
X = Indeterminado

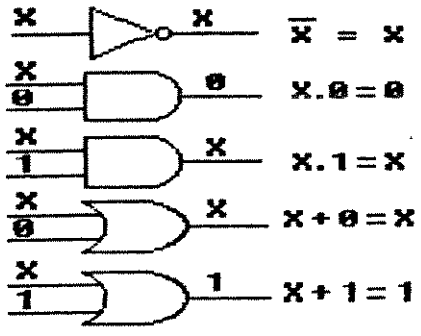


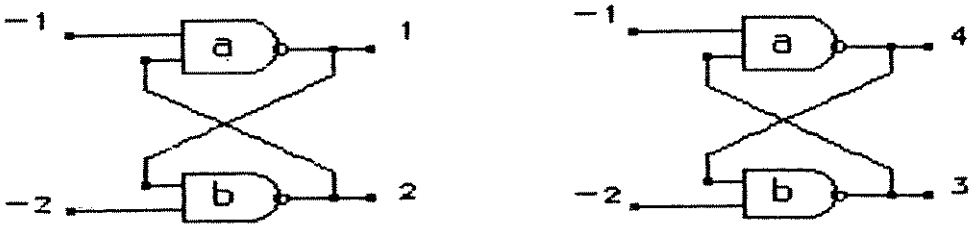
Tabela Verdade

	OR	AND
00	0	0
01	1	0
0X	X	0
10	1	0
11	1	1
1X	1	X
X0	X	0
X1	1	X
XX	X	X

Figura V.3.- Lógica de 3 valores

V.2.3.- CIRCUITOS SEQUENCIAIS

Na análise de circuitos sequenciais é fundamental o estudo da célula básica de memória definida na figura V.4.



NAND2(-1,2,1)
NAND2(-2,1,2)

NAND2(-2,4,3)
NAND2(-1,3,4)

Figura V.4.- Células básicas de memória.

Como o algoritmo não processa as duas portas simultanemente é possível observar-se que a resposta, dada uma entrada, poderia não ser estável. Assim decidiu-se pela inicialização de todas as saídas com o valor indeterminado. Numerosos testes validaram esta heurística.

A figura V.4 ilustra a chamada das rotinas das células básicas de memória, de duas maneiras, invertendo-se a ordem de chamadas das subrotinas

dos dois NANDs . Assim, o NAND (a) quando chamado primeiro tem a saída 1 e quando chamado em segundo lugar tem a saída 4. Do mesmo modo, o NAND (b) tem as saídas 3 e 2 respectivamente quando chamado em primeiro e segundo lugar. O resultado da simulação apresentado na figura V.5 comprova que as respostas são exatamente as mesmas nos dois casos simulados.

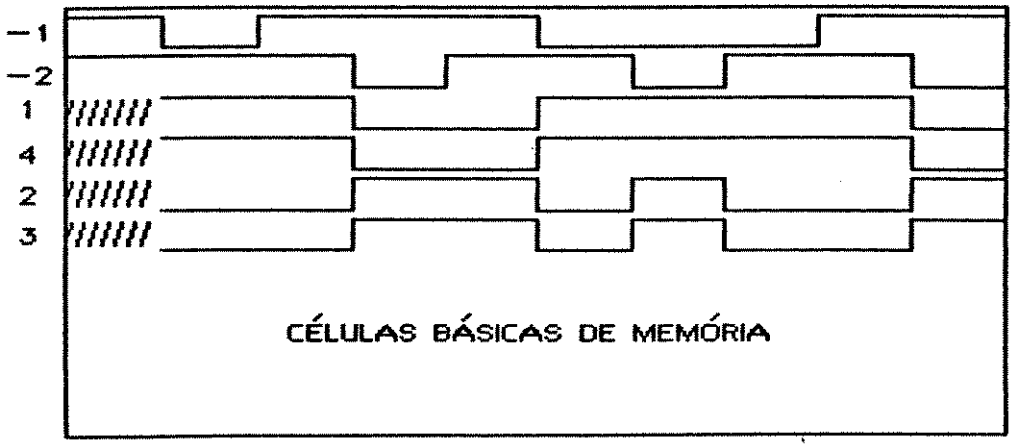


Figura V.5.- Resultado da simulação das células da figura V.4.

Na maior parte das vezes, um circuito sequencial consiste num arranjo de várias células básicas de memória do tipo mostrado na figura V.4 .

Desta forma, é possível determinar-se corretamente os estados do circuito para todo circuito sequencial constituido por células do tipo mostrado na figura V.4. é importante ressaltar que numerosos exemplos simulados validaram esta estratégia de simulação.

Para ilustrar o uso de Flip-Flops, criou-se rotinas para simular os flip-flops: 7474 que é do tipo D e 7473 que é do tipo J-K cujos esquemas são dados nas figuras V.6 e V.7, respectivamente. Estes esquemas de flip-flops foram obtidos de manuais de fabricantes de circuitos lógicos. As convenções de chamada de suas rotinas estão na figura V.8.

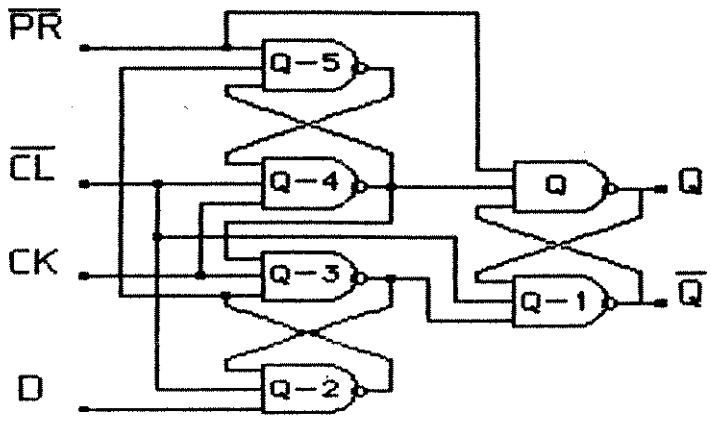


Figura V.6.- Flip-Flop D.

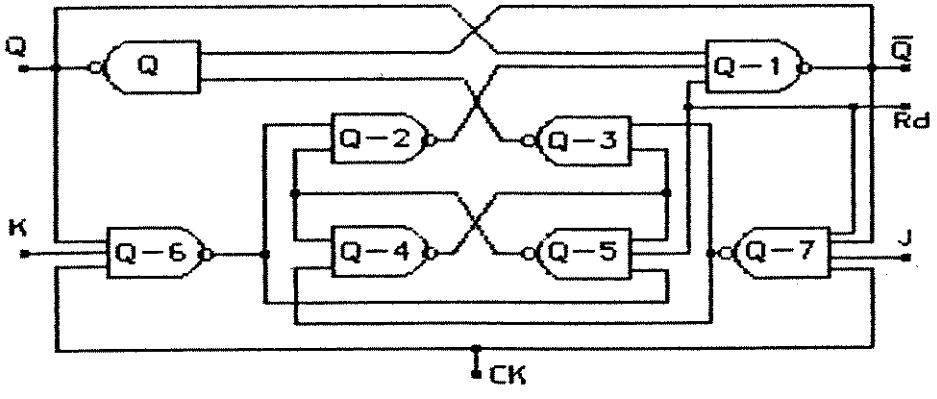


Figura V.7.- Flip-Flop JK.

```
PROCEDURE FF7474S6(D,Ck,Cl,Pr,Q:INTEGER);
   BEGIN
     NAND3(Q-2,Ck,Q-4,Q-3);
     NAND3(Q-5,Cl,Ck,Q-4);
     NAND3(Q-3,Cl,Q,Q-1);
     NAND3(Q-4,Pr,Q-1,Q);
     NAND3(Q-2,Pr,Q-4,Q-5);
     NAND3(D,Cl,Q-3,Q-2);
    END;
PROCEDURE FF7473S8(J,K,CK,RD,Q:INTEGER);
   BEGIN
     NAND4(J,Q-1,CK,RD,Q-7);
     NAND3(K,CK,Q,Q-6);
     NAND3(RD,Q-6,Q-4,Q-5);
     NAND2(Q-5,Q-7,Q-4);
     NAND2(Q-4,Q-7,Q-3);
     NAND2(Q-5,Q-6,Q-2);
     NAND3(RD,Q,Q-2,Q-1);
     NAND2(Q-1,Q-3,Q);
   END;
```

Figura V.8.- Rotinas para os Flip-Flops D fig. V.6 e JK fig. V.7.

V.3.-EXEMPLOS

Foram simulados alguns circuitos combinacionais associando a cada porta lógica um tempo de atraso de 10 ns tanto para TPLH como para TPHL.

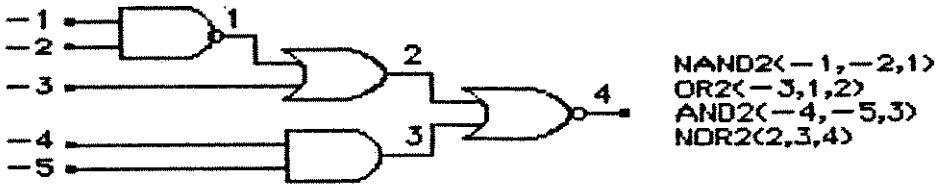


Figura V.9.- Exemplo de Circuito Combinacional.

A figura V.10 mostra o resultado da simulação do circuito da figura V.9 onde as entradas estão representadas por índices negativos e as saídas por positivos. Note que os instantes estão em microssegundos.

TRANSICAO	INSTANTE	-5	-4	-3	-2	-1	1	2	3	4
ENTRADA	0.000	0	0	0	0	0	X	X	X	X
INTERNA	0.010	0	0	0	0	0	1	X	X	X
INTERNA	0.010	0	0	0	0	0	1	X	0	X
INTERNA	0.020	0	0	0	0	0	1	1	0	X
INTERNA	0.020	0	0	0	0	0	1	1	0	X
INTERNA	0.020	0	0	0	0	0	1	1	0	X
INTERNA	0.030	0	0	0	0	0	1	1	0	0
INTERNA	0.030	0	0	0	0	0	1	1	0	0
INTERNA	0.030	0	0	0	0	0	1	1	0	0
ENTRADA	1.000	1	0	0	0	0	1	1	0	0
ENTRADA	2.000	0	1	0	0	0	1	1	0	0
ENTRADA	3.000	0	0	1	0	0	1	1	0	0
ENTRADA	4.000	0	0	0	1	0	1	1	0	0
ENTRADA	5.000	0	0	0	0	1	1	1	0	0
ENTRADA	6.000	1	1	1	1	1	1	1	0	0
INTERNA	6.010	1	1	1	1	1	0	1	0	0
INTERNA	6.010	1	1	1	1	1	0	1	1	0
INTERNA	6.020	1	1	1	1	1	0	1	1	0
ENTRADA	7.000	1	1	1	1	0	0	1	1	0
INTERNA	7.010	1	1	1	1	0	1	1	1	0
ENTRADA	8.000	1	1	1	0	1	1	1	1	0
ENTRADA	9.000	1	1	0	1	1	1	1	1	0
INTERNA	9.010	1	1	0	1	1	0	1	1	0
INTERNA	9.020	1	1	0	1	1	0	0	1	0
ENTRADA	10.000	1	0	1	1	1	0	0	1	0
INTERNA	10.010	1	0	1	1	1	0	1	1	0
INTERNA	10.010	1	0	1	1	1	0	1	0	0
INTERNA	10.020	1	0	1	1	1	0	1	0	0
ENTRADA	11.000	0	1	1	1	1	0	1	0	0
FINAL	12.000	0	1	1	1	1	0	1	0	0

Figura V.10.- Tabela Verdade para um circuito combinacional simulado com tempo de atraso de 10 ns.

Para exemplificar a simulação de circuitos sequenciais foi escolhido o FF 7474 (Fig. V.6). Como consequência do fato do simulador ser estrutural, não se especificam tempos de atraso para o flip-flop, mas estes são determinados pela propagação dos tempos de atraso de cada porta que o constitui.

As figuras V.11 a V.14 mostram a resposta gráfica do simulador aos testes para TPLH de SET (entrada Pr), TPHL de RESET (entrada Cl) e TPLH e TPHL de CLOCK (entrada Ck) para o FF N74LS74 descrito na figura V.6. Os resultados estão conformes às especificações dos fabricantes. Observe o efeito "zoom" nestas figuras, que permite uma observação detalhada de qualquer intervalo de tempo desejado.

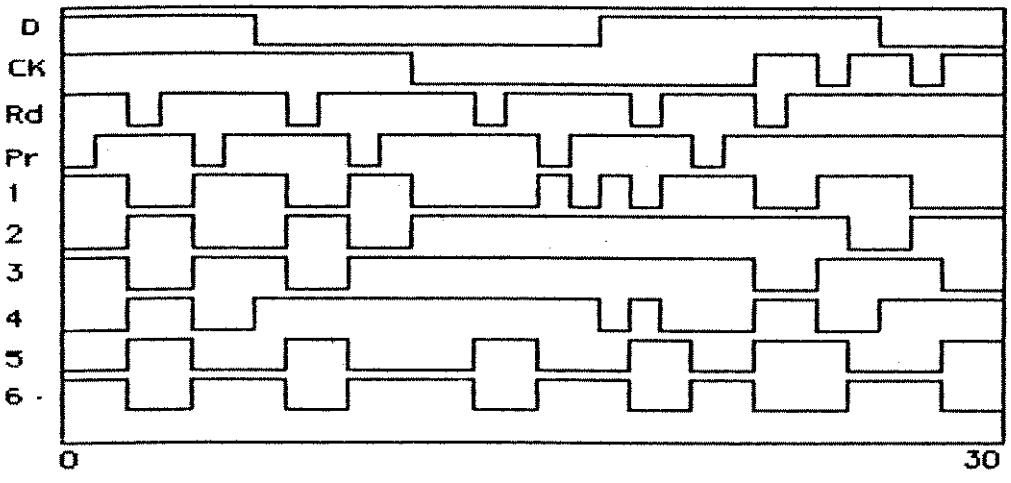


Figura V.11.- Diagrama de tempo para o FF D da fig.V.6.

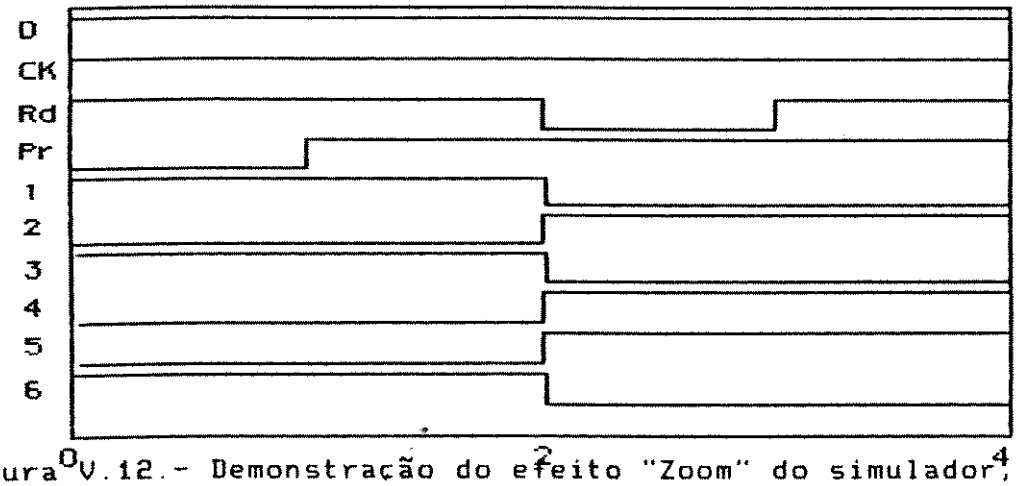


Figura V.12.- Demonstração do efeito "Zoom" do simulador, com amodificação do intervalo de tempo para ampliação das transições e melhor observação de detalhes.

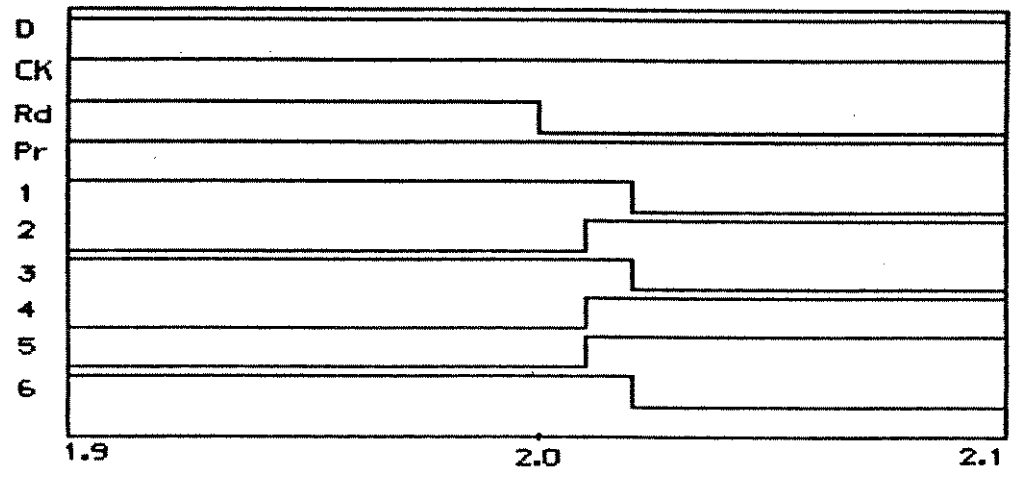


Figura V.13.- Novo "zoom", no entorno de 2.00 microssegundos para uma observação dos tempos de atraso nas transições.

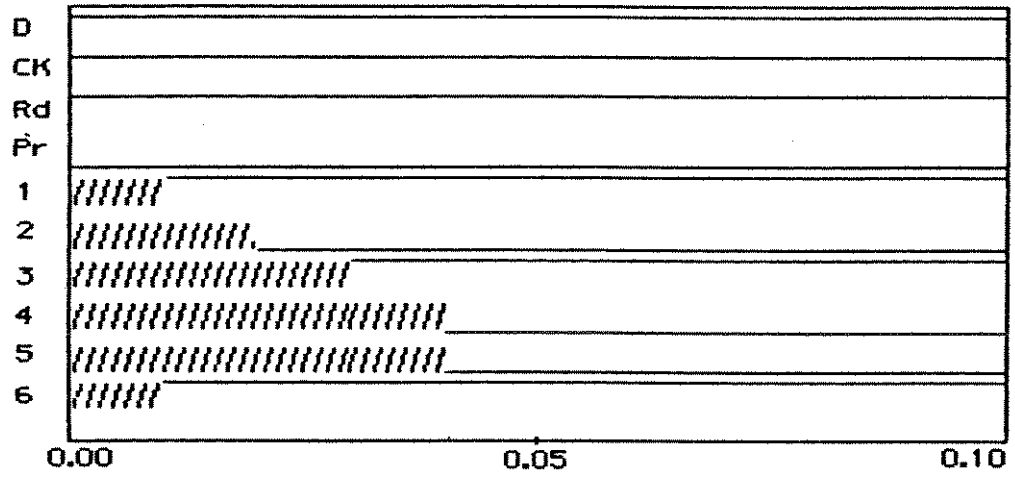


Figura V.14.- Ampliação do diagrama de tempo próximo ao instante zero para mostrar a representação gráfica do estado indeterminado "X" com o qual se inicializa o simulador.

O circuito de controle de sincronismo CCS [2] foi simulado usando-se o LÓGICO para ilustrar um exemplo um pouco mais complexo de circuito e a utilização e chamada das rotinas da biblioteca.

Através da simulação verificou-se que seu funcionamento é exatamente

aquele descrito nas tabelas e nos diagramas de estados correspondentes, não sendo detectadas falhas de projeto.

Através da simulação verificou-se que seu funcionamento é exatamente aquele descrito nas tabelas e nos diagramas de estados correspondentes, não sendo detetadas falhas de projeto.

As figuras V.15 e V.16 mostram a descrição do circuito para o simulador. A figura V.17 é o diagrama de tempo obtido da simulação de um ciclo completo do detector de sincronismo incluindo a perda e a recuperação do sincronismo.

```
NAND2(13, 31, 6)
NAND2(13, X, 5)
NAND2(X, 25, 4)
NAND3(6, 5, 4, 3)
NAND2(31, 19, 2)
NAND2(19, X, 1)
NAND3(2, 1, 4, 7)
FF7474S6(X, CK, /0, /0, 13)
FF7474S6(3, CK, /0, /0, 19)
FF7474S6(7, CK, /0, /0, 25)
FF7474S6(25, CK, /0, /0, 31)
```

Figura V.15.- Chamada das rotinas da biblioteca que formam o CCS.

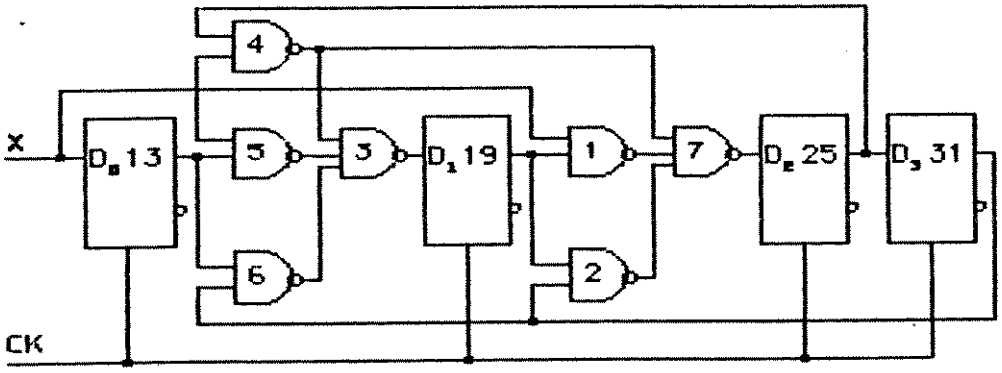


Figura V.16.- Descrição do CCS para o simulador LÓGICO.

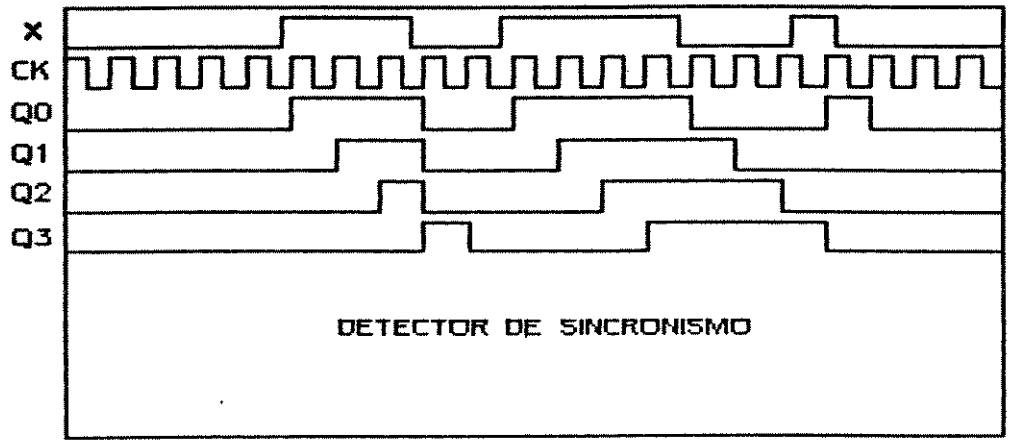


Figura V.17.- Diagrama de tempo obtido para o CCS.

Como um outro exemplo considere o circuito da figura V.18 onde é apresentado um flip-flop tipo D mestre-escravo construído com a tecnologia CMOS utilizando gates de transmissão (TG).

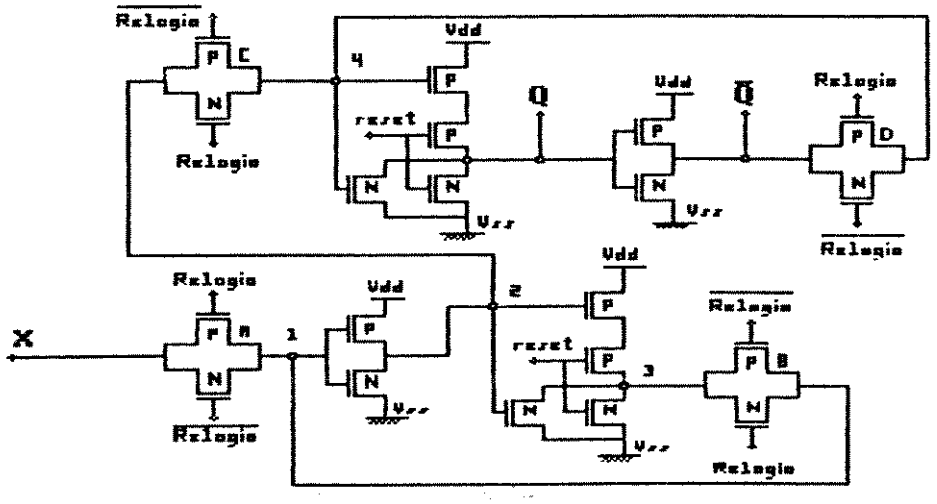


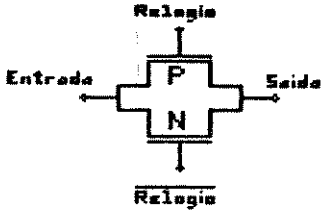
Figura V.18.- Flip-flop D mestre-escravo construido com a tecnologia CMOS utilizando gates de transmissão.

O gate de transmissão é uma chave analógica com dois estados, podendo ser levada a um estado de alta ou baixa impedância entre seus terminais.

Estes elementos podem ser utilizados como multiplexadores de sinais

onde suas saídas são conectadas num mesmo barramento e através de um sinal de contrôle estes gates são habilitados em instantes de tempo distintos.

A figura V.19 apresenta o esquema de um gate de transmissão utilizando a tecnologia CMOS e sua tabela de transmissão.



Gate de Transmissão CMOS

Transi.	conduz	abre
N	1	0
P	0	1

Figura V.19.- Gate de transmissão utilizando-se tecnologia CMOS e sua tabela de condução.

Um sinal de nível baixo aplicado ao elemento N causa uma alta resistência entre seus terminais (entrada/saída). Quando o nível deste mesmo elemento aumenta, sua resistência diminui e ele passa a conduzir (o nível de entrada transfere-se para a saída).

No circuito da figura V.18 a informação na entrada "D" é enviada para o mestre quando o relógio está em nível baixo. Quando o relógio vai para o nível alto a informação é enviada para o escravo. Esse processo é descrito a seguir:

1. Quando o relógio está baixo o gate de transmissão "A" conduz levando o nó "1" ao mesmo nível da entrada; enquanto que o gate de transmissão "B" permanece aberto (sem condução). Do mesmo modo o gate de transmissão "C" abre e o gate de transmissão "D" conduz, realimentando o nó 4.

2. Quando o relógio vai a nível alto o TG "A" abre e o TG "B" conduz realimentando o nó "2". O TG "C" conduz passando o estado do nó "2" para o escravo, enquanto que o TG "D" abre o laço de realimentação.

Um nível alto nas entradas de reset levará os nós "3" e "Q" ao nível baixo inicializando o flip-flop num nível de sinal baixo.

O circuito da figura V.18 pode ser interpretado funcionalmente através do modelo apresentado na figura V.20.

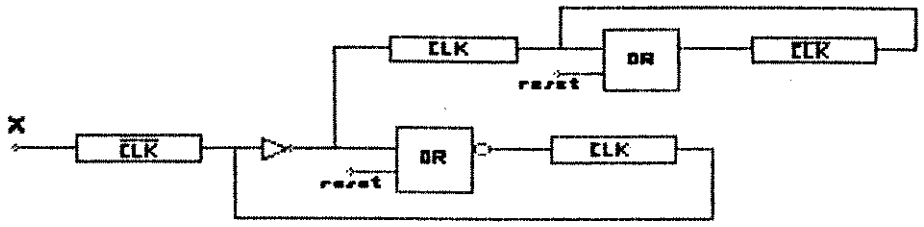


Figura V.20.- Modelo funcional do circuito da figura V.18.

A função de um gate de transmissão, pode ser executada através de gates primitivos como ilustrado pela figura V.21.

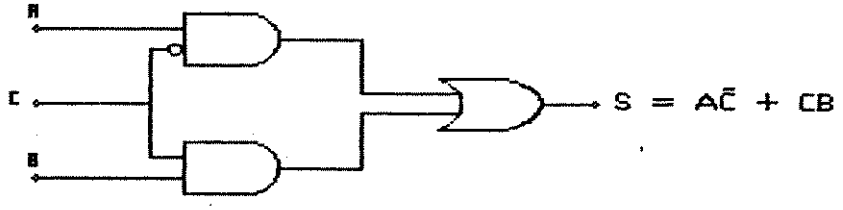


Figura V.21.- Função do gate de transmissão implementado com portas lógicas.

Deste modo o circuito da figura V.18 pode ser representado pelo circuito lógico apresentado na figura V.22.

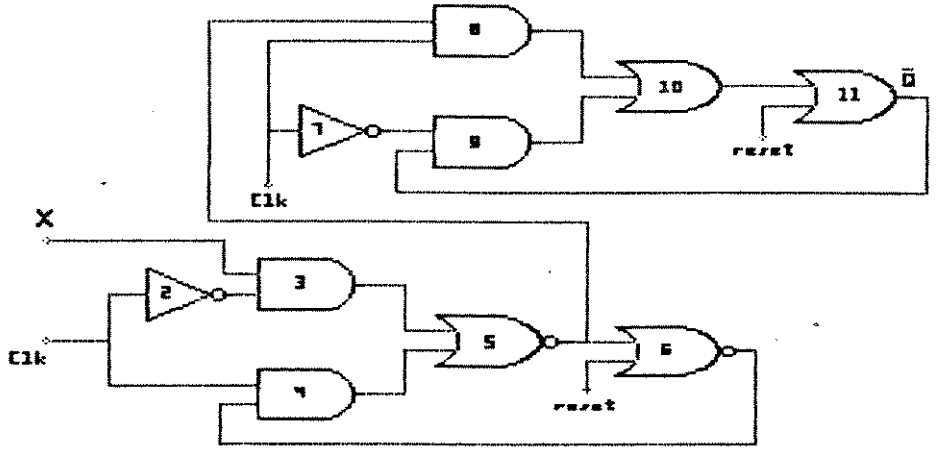


Figura V.22.- Circuito lógico equivalente ao circuito da figura V.18.

A figura V.23 apresenta o diagrama de tempo obtido pelo lógico.

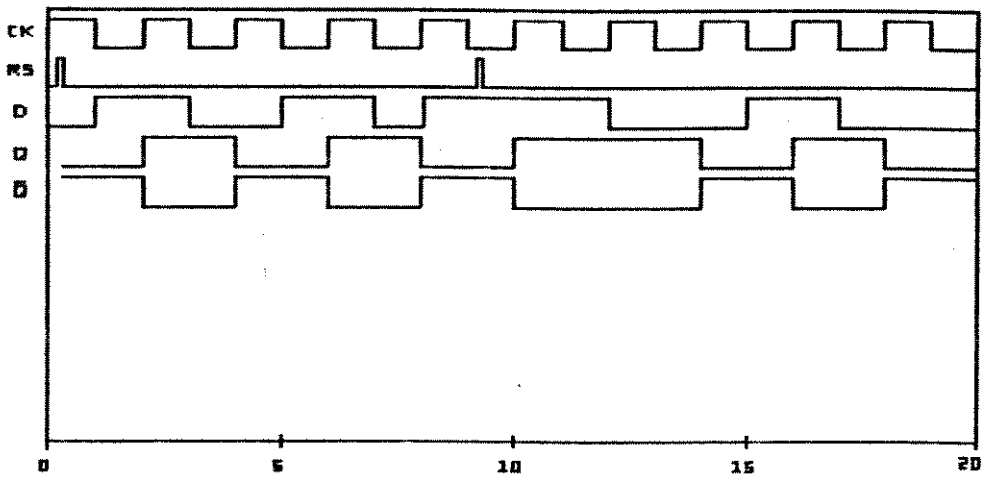


Figura V.23.- Diagrama de tempo do circuito da figura V.18 obtido pelo lógico.

V.4.- CONCLUSÃO

O maior detalhamento dado neste capítulo à técnica (simulação por eventos discretos) e modelo (lógica pseudo-ternária: 0,1 e X) para simulação no nível lógico reflete o estado atual do projeto assistido por computador. Apenas muito recentemente a simulação em níveis mais abstratos tem sido incorporada a produtos comerciais, e mesmo assim de forma bastante tênue, na forma designada por behavioral modeling pelos fabricantes, e que no mais das vezes corresponde simplesmente à possibilidade do usuário definir seus próprios módulos através de uma linguagem de programação de alto nível como C ou Pascal.

A simulação lógica ainda domina portanto o cenário, e a situação atual apresenta o grande inconveniente da ineficiência, por exemplo: A simulação lógica num PC/AT permite no máximo cerca de 500 avaliações/segundo, para circuitos com até 5000 portas [7].

O simulador LóGICO é útil no estudo dos circuitos lógicos, pois permite a obtenção dos diagramas de tempo de uma maneira simples poupando ao projetista um esforço considerável.

Graças às facilidades gráficas, pode-se visualizar diagramas de tempo dos circuitos simulados, escolhendo-se as ondas a serem estudadas e os intervalos de tempo desejados. Isto permite comparações entre diversas ondas e um efeito do tipo "zoom", aumentando-se ou diminuindo-se os tamanhos das janelas de tempo mostradas.

Outra facilidade implementada é a possibilidade de escolha de diferentes valores de tempos de atraso para os elementos do circuito, existindo inclusive a opção da geração de valores aleatórios entre limites máximos e mínimos fornecidos.

Devido a seu baixo custo sua utilização é conveniente na educação e para pequenos circuitos pois suporta no máximo 3000 portas lógicas.

V.5.- REFERÊNCIAS BIBLIOGRÁFICAS

[1] F. R. Wagner, I. Jansch-Porto, R.F. Weber, T.S. Weber
Métodos de Validação de Sistemas Digitais
IV Escola de Computação - Campinas, SP - 1988

[2] M. C. G. Madureira
Contribuição à Análise e Síntese de Circuitos Digitais
Tese de Mestrado - FEE-UNICAMP, Maio de 1987

[3] R. D. Chamberlain, M. A. Franklin
Collecting Data About Logic Simulation
IEEE Trans. on Computer Aided Design, CAD 5, No 3, July 1986

[4] E. P. Stabler
System Description Languages
IEEE Trans. on Computers, Vol. C-19, No. 12, December 1970

[5] G. Fantauzzi
An Algebraic Model for the Analysis of Logical Circuits
IEEE Trans. on Computers, Vol. C-23, No. 6, June 1974

[6] I.S. Bonatti, M.C.G. Madureira
Introdução à Análise e Síntese de Circuitos Lógicos
Editôra Unicamp, no prelo

[7] Goering, R.
Special Report on Design Automation: Simulation Challenges Breadbording for Design Verification
Computer Design, june 1986

C O N C L U S Ã O

Este trabalho de tese tratou da síntese, análise e simplificação de circuitos lógicos.

No capítulo II foram apresentados os conceitos básicos da álgebra booleana e as notações utilizadas em todo o trabalho.

As funções booleanas foram definidas afim de descrever as propriedades dos circuitos combinacionais.

A apresentação da secção II.4 (detecção de falhas) forneceu ao leitor a base necessária para o entendimento do método de minimização através da geração de padrão de teste introduzido no capítulo III.

A minimização de funções booleanas com grande número de variáveis permanece um problema importante devido ao uso extensivo de PLAs (Estruturas Lógicas Programáveis) no projeto de circuitos digitais.

O enfoque clássico na minimização consiste na obtenção de todos os implicantes primos e na seleção dos implicantes primos essenciais que realizam a cobertura mínima da função.

Tais métodos não são úteis acima de uma dezena de variáveis, pois a geração dos implicantes primos exige grande tempo computacional e grande espaço de memória e a cobertura em geral consome mais tempo ainda.

Portanto é interessante o estudo de métodos cobertura irredundante com o objetivo de reduzir o uso de memória e tempo computacional.

O método de Cobertura Irredundante Através da Geração de Padrão de Teste, desenvolvido neste trabalho, apresentou desempenho satisfatório quando comparado com alguns métodos da literatura. Seu grande potencial é baseado no uso restrito de memória.

No capítulo IV foi apresentada uma nova versão do programa TABELA, melhorada em seus aspectos computacionais, que realiza o projeto sistemático de máquinas sequenciais síncronas.

Este programa retira do projetista as tarefas cansativas e repetitivas de testar e realizar diferentes alocações de estado no projeto de uma dada máquina, permitindo concentrar-se na análise crítica das diferentes soluções.

Uma nova versão do simulador LóGICO foi apresentada no capítulo V, melhorando-se seus aspectos computacionais, resultando em um instrumento útil no estudo dos circuitos lógicos através da obtenção dos diagramas de tempo de

uma maneira simples e eficiente.

Graças às facilidades gráficas, pode-se visualizar diagramas de tempo dos circuitos simulados, escolhendo-se as ondas a serem estudadas e os intervalos de tempo desejados. Isto permite comparações entre diversas ondas e um efeito do tipo "zoom", aumentando-se ou diminuindo-se os tamanhos das janelas de tempo mostradas.

Outra facilidade implementada é a possibilidade de escolha de diferentes valores de tempos de atraso para os elementos do circuito, existindo inclusive a opção da geração de valores aleatórios entre limites máximos e mínimos fornecidos.

Por fim é necessário explicitar que a geração e manutenção de programas computacionais de apoio ao projeto de circuitos lógicos é uma tarefa permanente. Nesse sentido este é um campo de trabalho sempre aberto à pesquisa e ao desenvolvimento de novos programas computacionais.