DISEÑO DE CIRCUITOS LÓGICOS ASISTIDO POR COMPUTADORA

UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

DISEÑO DE CIRCUITOS LÓGICOS ASISTIDO POR COMPUTADORA

MIGUEL EDUARDO GÓMEZ GORDILLO

Trabajo de investigación presentado para
optar al grado académico de
Licenciatura en Ingeniería Electrónica

Guatemala

1992

Vo. Bo. :

(f) _____
Ingeniero Ricardo Cordón
Asesor

Tribunal:

(f) _____
Ingeniero Ricardo Cordón

(f) _____
Licenciado Roberto Tejada

(f) _____
Ingeniero Rolando Mata

Fecha de aprobación: 9 de septiembre de 1992.

A Dios y

a mis padres

CONTENIDO

Resumen

I. INTRODUCCIÓN 1

A. Funciones de conmutación 1

II. FUNDAMENTACIÓN TEÓRICA 3

A. Número de funciones de conmutación 3
B. Minimización de funciones de conmutación 4
C. Métodos de minimización 5
1. Minimización manual. 6
2. Método de Quine-McClusky. 7

III. SÍNTESIS DE CIRCUITOS LÓGICOS 9

A. Diseño automático - métodos y problemas 10
B. Requerimientos 12

IV. IMPLEMENTACIÓN EN C++ 15

A. Programación orientada hacia objetos 15
B. Clase Term 16
C. Almacenamiento 18

V. PROGRAMA DE DISEÑO LÓGICO 21

A. Descripción del programa 21
1. Ingresar nueva tabla de verdad. 22
2. Modificar tabla de verdad. 22
3. Ver lista de términos. 22
4. Ver lista de primeros implicados. 22
5. Ver lista de 1´s. 23
6. Calcular primeros implicados. 23
7. Minimización. 23
B. Algoritmo general 24
C. Limitaciones 25

VI. RUTINAS PRINCIPALES DEL PROGRAMA 27

A. Rutinas de la clase term 28
1. void term::unos(). 28
2. void term::representados(int,int). 30

B. Rutinas varias 30
1. void ordena(term,int). 31
2. int bits_dif(term,term). 31
3. void condensa(). 31
4. void elimrep(term,int,term,int). 34
5. void sacaprim(). 34
6. void primeros_implicados(). 34
7. void quita_repre(term,term,int). 35
8. void SubPrimImp(). 35

VII. CONCLUSIONES 37

VIII. BIBLIOGRAFÍA 39

RESUMEN

El objetivo de este trabajo es proporcionar una herramienta de software para la elaboración y diseño de circuitos electrónicos digitales. La función principal de este software es automatizar el proceso de diseño de circuitos lógicos en los procesos donde se requiere una gran cantidad de cómputos y que, por lo tanto, están expuestos a errores.

El desarrollo de un programa de esta naturaleza requiere características especiales en el lenguaje que será empleado. El lenguaje escogido fue C++, el cual, además de tener las características propias del lenguaje C, también posee características de la programación orientada hacia objetos que resultan muy útiles para resolver los problemas de representación de los distintos elementos usados en los algoritmos de conmutación.

Existen varios métodos para el diseño de circuitos óptimos, según distintos criterios. El criterio que se tomó en cuenta fue el del número de compuertas lógicas necesarias para la implementación de una función de conmutación dada. Específicamente se utilizó el método de Quine-McClusky para la minimización de funciones de conmutación. Fue necesario hacer ciertas modificaciones al método para incorporar el diseño de circuitos de múltiples salidas.

El programa presenta los resultados en forma de términos binarios. A partir de estos términos se puede obtener la

función de conmutación minimizada, por medio de la cual puede obtenerse a su vez el diagrama de circuito. Este diagrama estaría presentado en forma de circuito de dos niveles, ya que la expresión minimizada aparece en forma de suma de productos normalizada.

I. INTRODUCCIÓN

Un circuito combinacional consiste en compuertas lógicas, cuyas salidas se determinan directamente, en cualquier momento, de la combinación presente de entradas sin tener en cuenta las entradas anteriores. El proceso de diseño de circuitos combinacionales empieza con el enunciado del problema y termina con un conjunto de funciones de Boole, de las cuales se puede obtener el diagrama lógico fácilmente. El procedimiento es el siguiente:

- Enunciar el problema.
- Determinar las variables de entrada y de salida.
- Deducir la tabla de verdad que define las relaciones entre las entradas y las salidas.
- Obtener la función de Boole simplificada para cada salida.
- Dibujar el diagrama lógico.

A. Funciones de conmutación

Una función de conmutación describe matemáticamente el comportamiento lógico de un circuito combinacional de conmutación con n terminales de entrada x1, x2, ..., xn y m terminales de salida z1, z2, ..., zm. No es lo mismo que una función booleana. Una FC toma los valores del conjunto

P2 = {0,1}, mientras que una función booleana los toma de cualquier conjunto P.

La especificación de una FC particular puede tomar una de varias formas. El álgebra booleana provee una de ellas:

$$F = A + BC'$$

Una evaluación de F para todas las posibles combinaciones de valores de A, B y C, provee los datos necesarios para una segunda representación de una FC, la tabla de verdad (Tabla 1).

Tabla 1
Tabla de verdad

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Una tercera forma de representación es el diagrama de circuito.

II. FUNDAMENTACIÓN TEÓRICA

A. Número de funciones de conmutación

¿Cuántas funciones de conmutación existen? Ya que cada una tiene una única tabla de verdad, sólo es necesario determinar cuántas tablas de verdad diferentes existen. Una tabla de verdad completa (Tabla 2) tiene 2^n filas, una por cada una de los 2^n símbolos de entrada de n variables independientes.

Tabla 2
Tabla de verdad completa

x_1	x_2	...	x_n	z
0	0	...	0	a_0
0	0	...	1	a_1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
1	1	1	1	a_{2^n-1}

a_d representa el valor de la variable dependiente para el símbolo de entrada d.

Cada FC está especificada por una columna diferente de variable dependiente, una 2^n-tupla (a_0 ... a_{2^n-1}) diferen-

te. ¿Cuántas 2^n-tuplas binarias existen? Esta cuenta es 2^{2^n}, donde n es el número de variables independientes. Entonces, existen 2^{2^n} funciones booleanas de n variables.

Tabla 3
Número de FC's

n	2^n	2^{2^n}
0	1	2
1	2	4
2	4	16
3	8	256
4	16	65536
5	32	$\sim 4.295 \times 10^9$
6	64	$\sim 1.845 \times 10^{19}$

El número de funciones crece muy rápido con n (Tabla 3).

B. Minimización de funciones de conmutación

Minimización[1] es el proceso para obtener aquella expresión de una función de conmutación que es óptima bajo cierto criterio. Un criterio usual es el costo. Este criterio puede ser simple o complejo. En un tiempo, el costo de los diodos era elevado en la construcción de compuertas AND y OR, y se

/1/El término minimización se refiere a la obtención de una expresión mínima para una función de conmutación.

menospreciaba el costo de resistencias y fabricación. Luego, los transistores eran caros y se despreciaba el costo de los diodos, por lo que era importante reducir el número de transistores a utilizar. Un caso diferente sería para el diseñador que utiliza IC's (circuitos integrados) ya existentes, para el cual lo importante es el número de IC's a utilizar.

Los IC's de compuertas presentan un nuevo problema. El costo para hacer la conexión a una compuerta o montar la compuerta en el material de soporte puede igualar o exceder el costo de la compuerta misma. El costo de diseño puede exceder al costo del hardware si se van a fabricar unos pocos sistemas. En este caso, la primera expresión que se diseñe podría ser la óptima, ya que requiere el menor esfuerzo de diseño.

C. Métodos de minimización

A partir de una tabla de verdad se obtienen los términos mínimos (minterms), que representan una determinada función de conmutación. Una función de conmutación representada con términos mínimos está en su forma canónica. En la forma canónica cada término contiene todas las variables (negadas o no), por lo que no es una forma práctica para sus repre-

sentación en circuito. Es necesario entonces minimizar esa ecuación canónica utilizando algún método de minimización.

1. Minimización manual. Los axiomas y teoremas del álgebra Booleana pueden utilizarse para minimizar una función de Boole. Sin embargo, esto tiene serias desventajas cuando no se trata de funciones simples. En primer lugar, ¿en qué orden se deben aplicar los axiomas y teoremas? No hay respuesta. Con experiencia, suerte y a prueba y error podríamos encontrar un circuito mínimo, pero desde luego sería deseable un procedimiento ordenado. En segundo lugar, ¿cuándo debe detenerse en aplicar los axiomas y teoremas para seguir minimizando? Con algunas excepciones, es difícil decir cuándo una función de Boole se ha expresado en forma óptima.

Un método para simplificación de funciones de conmutación es el llamado Mapas de Karnaugh. Este método es conveniente para funciones con pocas variables de entrada, cinco o seis como máximo. Una de las principales desventajas de este método es el procedimiento de prueba y error que depende de la habilidad del diseñador para reconocer ciertos patrones. En funciones de seis variables por ejemplo, es difícil saber si realmente se obtuvo la expresión más simplificada.

2. Método de Quine-McClusky. Otro método de minimización es el llamado Método Tabular. Este fue formulado primero por Quine y más tarde mejorado por McClusky, por lo que es también llamado el método de Quine-McClusky. Es un procedimiento específico paso a paso que se puede aplicar a problemas con muchas variables. Básicamente, este método consiste en dos etapas. La primera es encontrar todos los términos candidatos de inclusión en la función simplificada. Estos términos se llaman los "primeros implicados". La segunda etapa es escoger un subconjunto de primeros implicados que dan una expresión con el menor número de literales.

El método de Quine-McClusky parte de una lista de términos mínimos que especifican por completo la función de conmutación. Sigue con un proceso de pareamiento, en el cual compara cada término mínimo con cada uno de los restante términos mínimos. Si dos términos mínimos difieren en una sola variable, esa variable se elimina para encontrar un término con una literal menos. Este proceso se repite para cada término mínimo de la lista inicial. Se obtiene entonces una nueva lista con términos "condensados" (una literal menos), la cual es procesada de la misma manera que la lista inicial. El ciclo se repite hasta que ya no se puedan condensar más términos, o sea, hasta que ya no se produzca una nueva lista. Los términos que no se condensaron durante el proceso constituyen los primeros implicados.

Después de esta primer etapa, viene la selección de un subconjunto de primeros implicados para minimizar lo más posible la expresión final. Esta selección puede realizarse de varias formas, siempre y cuando el subconjunto obtenido represente por completo la función inicial. Esto quiere decir que el subconjunto de primeros implicados debe representar todos y cada uno de los términos mínimos iniciales.

III. SÍNTESIS DE CIRCUITOS LÓGICOS

La síntesis de circuitos lógicos es el proceso de automatización de tareas, tradicionalmente laboriosas, para traducir un concepto de diseño a una implementación a nivel de compuertas lógicas. El proceso incluye la optimización de resultados con base en una tecnología específica para mejorar el rendimiento y/o reducir el área de circuito.

Esta herramienta de diseño estuvo disponible comercialmente a finales de los años 80 y su aceptación por diseñadores profesionales ha sido inmediata. Algunos aspectos importantes son:

- Ambos fabricantes de sistemas y proveedores han adoptado la síntesis lógica como parte de las herramientas básicas para diseñadores. Empresas importantes en electrónica y computación han realizado compras considerables de software de síntesis lógica para uso interno en sus departamentos de ingeniería.

- La síntesis lógica ha jugado un papel fundamental en el diseño exitoso de productos de alta calidad. En Sun Microsystems la síntesis lógica fue indispensable para el diseño de dos chips LSI (integración a gran escala), que hicieron posible la introducción de los sistemas de gráficas para estaciones de trabajo.

- Empresas líderes en fabricación de chips, incluyendo a Fujitsu, el proveedor más grande de arreglos de compuertas del mundo, y LSI Logic, están poniendo a la disposición de sus clientes la síntesis lógica como herramienta central en sus procesos de automatización.

- Evaluaciones de compañías líderes en alta tecnologia han demostrado un incremento del 30% en velocidad y 24% de reducción en área de circuitos. Esto se ha logrado a través de síntesis lógica.

Es una regla general en la industria el hecho de que los ingenieros de diseño invierten 80% de su tiempo rediseñando circuitos ya existentes y el 20% diseñando nuevos circuitos. Las herramientas de síntesis lógica pueden llevar a cabo ambas actividades.

A. Diseño automático = métodos y problemas

La incorporación de un algoritmo de conmutación (switching) como el de McCluskey en un programa de computadora no es fácil. Al escribir estos programas se trabaja especialmente en la evaluación y manipulación de bits y

patrones de bits en lugar de números. A pesar de que la base de operación de la computadora es con patrones de bits, sus lenguajes principales como Algol, Fortran, Basic, etc., están sin lugar a dudas orientados hacia el proceso de datos numéricos. Más aún, algunas computadoras están organizadas en una base decimal, lo que las hace inútiles para este tipo de trabajo.

Es necesario utilizar una máquina binaria si se desea lograr cierto grado de eficiencia en los programas de conmutación. Por ello, el desarrollo debe ser en lenguaje de máquina básico o alguna forma de programa ensamblador (assembler) simbólico, con tal de realizar las operaciones de bits requeridas, y también para ejercer control sobre las características de espacio y velocidad de los programas. En algunos casos, un lenguaje de alto nivel puede ser usado para la organización del programa, sirviéndose de subrutinas en lenguaje de máquina para realizar las operaciones de bits y el proceso de los patrones de bits. Aun así, las instrucciones básicas de muchas computadoras son inadecuadas para el propósito. Por ejemplo, pocas computadoras incluyen operaciones lógicas, excepto por la función AND. Consecuentemente es necesario recurrir a la elaboración de subrutinas para las operaciones lógicas, lo cual da como resultado pérdidas en tiempo y espacio de almacenamiento.

Otra área en la que la computadora es inadecuada para programas de conmutación es en el almacenamiento de datos. Los términos de conmutación son almacenados normalmente en notación binaria usando una representación posicional; por ejemplo a´b´c´d se almacena como 0001 en alguna palabra de computadora. En la mayoría de problemas, sin embargo, es también necesario almacenar otra información como: número de función del término y variables eliminadas.

B. Requerimientos

Parece ser que en su forma presente, la estructura básica y lenguajes de las computadora digitales no son adecuados para problemas de conmutación o no-numéricos. Esto quiere decir que la programación de problemas de conmutación en particular, resulta en programas ineficientes usando grandes cantidades de memoria y requiriendo tiempos largos de proceso. Además, escribir programas en lenguaje de máquina es por sí mismo difícil. Por ello, el punto de vista de muchos autores es que los problemas de conmutación están fuera del alcance de las computadoras digitales.

Sin embargo, muchos autores sostienen que la aproximación correcta hacia el diseño de sistemas de computadora es empezando con los lenguajes orientados hacia problemas

específicos. Para ello es necesario el desarrollo de dos distintos tipos de lenguaje de computadora, uno para cómputos numéricos y otra para aplicaciones no-numéricas.

La programación de aplicaciones no-numéricas está primeramente relacionada con la manipulación de bits y de símbolos, particularmente en ordenamiento, análisis, modificación, etc., de listas o tablas. Las características principales de tal lenguaje son:

1. Manipulación de símbolos que no tienen significado numérico.
2. Los requerimientos de almacenamiento no pueden ser especificados con anticipación. Por ejemplo, en un problema de diseño lógico es imposible determinar, a partir de las ecuaciones de entrada, cuántos espacios de almacenamiento serán necesarios para el proceso de minimización.
3. La organización de espacio y las relaciones entre los datos cambian constantemente durante la ejecución del programa.
4. Son necesarias modificaciones frecuentes de los datos y el programa mismo.

IV. IMPLEMENTACIÓN EN C++

Para el desarrollo del programa de diseño lógico en cuestión, se utilizó el lenguaje C. Específicamente se trabajó con el C++ de Borland, que permite el uso de "clases".

A. Programación orientada hacia objetos

La programación orientada hacia objetos es un método de programación que busca imitar la forma humana de conceptualizar el mundo. Se utiliza en este método el concepto de "clases". De hecho, el lenguaje C++ fue originalmente llamado C con clases. El uso de clases facilita la implementación en un lenguaje de computadora de los algoritmos para diseño lógico, ya que se pueden manipular objetos individuales y realizar operaciones no numéricas con ellos.

Una característica importante de la programación orientada hacia objetos es la forma en que se manejan los datos y las funciones. En los lenguajes tradicionales no existe una relación explícita entre datos y funciones, lo que implica que cualquier programador puede utilizar los datos de las estructuras sin necesidad de utilizar las funciones dadas. En C++, una clase combina funciones y datos de tal manera que ambos elementos son implícitos a esa clase.

B. Clase Term

Para representar un término binario se definió una clase llamada "term", la cual tiene, en el lenguaje C++, la siguiente estructura:

```
class term {
    private:
    int valor;    // Valor decimal del término.
    int x;        // Posición de los don't cares.
    int func;     // Función de salida.
    public:
    int leeval(); // Para obtener valor.
    int leex();   // Para obtener x.
    int leefun(); // Para obtener func.
    int unos();   // Número de unos en un término.
    void asigna(int elvalor=0, int lax=0, int
lafunc=0);
        void representados(int *vector, int
&nvector);
}
```

La clase term se compone de datos y funciones. Los datos son: valor, x y func. Para representar un término se especi-

fica cada uno de estos datos ({valor,x,func}). Las funciones de la clase term son las operaciones implícitas a esta clase. Esto es, las operaciones que operan exclusivamente sobre elementos de clase term.

Como ejemplo analicemos el término binario

1x01.

Representado en clase term es como sigue:

{9,4,1}

valor = 9 ... valor decimal del número binario 1001.

x = 4 ... ubicación del don't care (0x00)

func = 1 ... pertenece a la función de salida 1.

Ahora bien, una de las funciones que pueden operarse sobre este término es la función unos(). En el siguiente fragmento de programa:

```
term ejemplo;
g = ejemplo.unos();
```

Si

ejemplo = {9,4,1}

entonces

g = 2

que es el número de unos que posee el término ejemplo (1x01).

Otro ejemplo,

111x.

Este término, en clase term, se representa como:

{14,1,1}

valor = 14 ... valor decimal del número binario 1110.

x = 1 ... ubicación del don't care (000x)

func = 1 ... pertenece a la función de salida 1.

El valor de la función unos() es 3, ya que el término posee tres unos.

C. Almacenamiento

Como se señaló anteriormente, el manejo de términos binarios es en forma decimal. Sin embargo, al almacenar un término en una posición de memoria determinada, éste es almacenado en forma binaria dependiendo del tipo de dato que se haya escogido para el término. En C se tienen los siguientes tipos de datos numéricos: int y long. Para la manipulación de los términos se escogió el tipo long, el cual es almacenado como una palabra de 16 bits.

La ventaja del lenguaje C respecto de las posiciones de memoria es que se tiene acceso directamente a cada bit de la palabra almacenada. Por ejemplo, el número 7, siendo de tipo long, es almacenado como 0000000000000111. Existen instrucciones propias del lenguaje para la manipulación de los bits en esta palabra. Algunas de estas instrucciones son: <<

(shift left), & (and), | (or), etc. Todo esto tiene la ventaja adicional de que ya no es necesario desarrollar una rutina especifica para conversión de decimal a binario, ya que el lenguaje lo hace implícitamente.

V. PROGRAMA DE DISEÑO LÓGICO

A. Descripción del programa

El programa de diseño lógico consiste básicamente de tres etapas: ingreso de datos, cálculo de primeros implicados y minimización. Una de las características principales del programa es que utiliza el método de Quine-McClusky modificado para varias salidas. El ingreso de datos se hace por medio de tablas de verdad de una o varias variables de entrada y salida. El resultado es un subconjunto de primeros implicados expresados en forma de términos de clase term (ver implementación en C++). El menú principal del programa es el siguiente:

1. Ingresar nueva tabla de verdad
2. Modificar tabla de verdad
3. Ver lista de términos
4. Ver lista de 1's
5. Ver lista de primeros implicados
6. Calcular primeros implicados
7. Minimización
0. Terminar e ir a DOS

Cada una de estas opciones se explica a continuación.

1. Ingresar nueva tabla de verdad. Por medio de esta opción se ingresa la tabla de verdad de la función de conmutación que se desea trabajar. Esta opción borra los datos ya existentes y permite el ingreso de una tabla totalmente nueva. Al escoger esta opción el programa solicita que se ingrese primero el tamaño deseado de la tabla, o sea, cuántas variables de entrada y salida se deberán tomar en cuenta, y luego muestra la tabla con 0´s en todas las funciones de salida.

2. Modificar tabla de verdad. Esta opción se utiliza para modificar los datos ya existentes de una tabla de verdad previamente ingresada. La opción permanece inválida mientras no se haya ingresado una tabla por medio de la opción 1. Al modificar una tabla existente no se borran los datos de la misma. El programa únicamente vuelve a mostrar la tabla existente en pantalla para poder ser modificada.

3. Ver lista de términos. Con esta opción es posible ver la lista de términos mínimos iniciales. Estos se muestran en forma de términos de clase term. Se muestran, tanto los 1´s, como las x´s.

4. Ver lista de primeros implicados. Esta opción se utiliza para ver la lista de primeros implicados luego de

que éstos fueron calculados. Al escoger esta opción, los primeros implicados no se vuelvén a calcular, sino que solamente se muestran en pantalla los primeros implicados existentes en memoria. Para utilizar esta opción fue necesario calcular, previamente, los primeros implicados.

5. Ver lista de 1´s. En algunas ocasiones es necesario ver los términos mínimos que hacen que la función sea 1. Se muestran solamente los 1´s. O sea que se ignoran los 0´s y las x´s.

6. Calcular primeros implicados. Esta opción se utiliza para realizar el cálculo de los primeros implicados. Cada vez que se escoge esta opción el programa vuelve a determinar los primeros implicados utilizando el método de Quine McClusky para múltiples salidas. El mismo método se utiliza, tanto para varias salidas, como para una sola.

7. Minimización. Una vez determinados los primeros implicados puede utilizarse esta opción para obtener un subconjunto de primeros implicados. El programa permite dos formas de obtener estos subconjuntos. La primera requiere que el usuario ingrese el número de veces que desea que el programa intente conseguir un subconjunto mínimo. Se obtiene, en forma aleatoria, un subconjunto. Si el subconjunto

obtenido es menor que el anterior, entonces lo muestra en pantalla, de lo contrario vuelve a repetir los cálculos. Este proceso lo repite el número de veces que haya ingresado el usuario. La otra forma de obtener un subconjunto de primeros implicados es mostrando el resultado cada vez que se logra un subconjunto. Las pruebas también se realizan en forma aleatoria, pero se muestra cada subconjunto obtenido.

B. Algoritmo general

A continuación se presenta el algoritmo general del programa de diseño lógico. Este algoritmo incluye únicamente la secuencia general del programa, es decir, no incluye los algoritmos específicos para cada rutina. Estos algoritmos específicos se encuentran en las descripciones de las rutinas correspondientes.

1. Ingresar términos (minterms) a un vector principal de minterms.
2. Ordenar el vector principal según el número de unos que contenga cada término.
3. Condensar el vector principal a vector de términos condensados y registrar los términos no condensados en un vector de primeros implicados.

4. Eliminar términos repetidos del nuevo vector de términos condensados.
5. Escoger ahora el nuevo vector de términos condensados como vector principal.

   ¿El vector principal tiene elementos?

   Sí -> 3

   No -> 6
6. Escoger el vector resultante de primeros implicados.
7. Sacar, al azar, un subconjunto de primeros implicados.
8. Mostrar el subconjunto de primeros implicados junto con el número de términos que contiene.
9. ¿Fin de búsqueda de subconjuntos?

   Sí -> 11

   No -> 8
10. Fin.

C. Limitaciones

El ingreso de datos se realiza únicamente a través de una tabla de verdad. Para circuitos de hasta 4 variables de entrada esto es suficiente. Sin embargo, para circuitos de 5 o más variables resulta un proceso largo el ingreso de la tabla de verdad.

Los resultados del programa se presentan en forma de términos de clase term. O sea que el programa produce prácticamente sólo circuitos de dos niveles (suma de productos). La conversión de esta forma de circuito a otras representaciones debe realizarla el usuario por separado.

Respecto de velocidad, depende en gran parte de la computadora en la que se opere el programa. Es de gran ayuda si la computadora posee coprocesador matemático, ya que de esta forma se acelera la rutina unos(), la cual es posiblemente la rutina más accesada por el programa. El desarrollo y prueba del programa se hizo en una computadora 386SX de 20 Mhz, y la respuesta en tiempo para 6 variables de entrada y 8 de salida es casi inmediata. Sin embargo, el programa resultaba bastante lento al realizar las operaciones en una computadora XT-8088 de 8Mhz. Con un coprocesador matemático en esta última computadora, el tiempo de proceso fue similar al de la 386SX de 20 Mhz.

La limitación principal del programa es respecto del espacio de almacenamiento en memoria RAM. Con 2Mb de memoria RAM el programa opera sin problemas hasta 6 variables de entrada y 8 de salida. Sin embargo, sólo es necesario aumentar la memoria para tener acceso a mayor número de variables.

VI. RUTINAS PRINCIPALES DEL PROGRAMA

Para la codificación del programa de diseño lógico se desarrollaron rutinas específicas para la manipulación de los patrones de bits. Las rutinas principales son:

- void term::unos()
- void term::representados(int,int)
- void ordena(term,int)
- int bits_dif(term,term)
- void condensa(term,term,int,int,int,int)
- void elimrep(term,int,term,int)
- void sacaprim(int,int,term,int,term,int)
- void primeros_implicados(term,int,term,int)
- void quita_repre(term,term,int)
- void SubPrimImp(term,int,term,int,term,int)

El código de todas estas rutinas se encuentra en los archivos FUNC001.CPP y FUNC002.CPP. Al momento de ser llamadas desde el programa principal únicamente es necesario asignarle los parámetros correspondientes. Estas rutinas pueden dividirse en dos tipos: pertenecientes a la clase term y no pertenecientes a la clase term.

A. Rutinas de la clase term

Estas rutinas pertenecen a la clase term. O sea que solamente pueden ser llamadas por elementos de esa clase.

1. void term::unos(). La rutina unos() es en realidad una función de la clase term y regresa el número de unos que contiene un término binario. Por ejemplo, el término 0001 contiene 1 uno, y el término 1011 contiene 3 unos. Para determinar si existe un 1 en determinada posición del término, la función hace un AND del valor del término con un número que contenga un 1 precisamente en esa posición. Si el resultado del AND es igual al número en comparación, entonces sí existe un 1 en la posición en cuestión. El proceso se repite hasta determinar si hay un 1 en cada posición de la palabra del término. Como ejemplo contemos cuántos 1's tiene el término binario 1011. Primero se hace un AND de 1011 con 0001.

1011 AND 0001 = 0001

Como el resultado es igual al número comparado (0001), quiere decir que sí es un 1. Luego se hace un AND del mismo término con un número que tenga un 1 en la siguiente posición de la palabra o sea con 0010.

1011 AND 0010 = 0010

Otra vez es igual, entonces quiere decir que hay otro uno. Sigue la comparación con un número que tenga un 1 en la siguiente posición, o sea con 0100.

1011 AND 0100 = 0000

El resultado no es igual al número comparado. Esto quiere decir que en esa posición no hay un 1. Finalmente se hace la operación

1011 AND 1000 = 1000

Nuevamente el resultado es el mismo que el número comparado, por lo que se cuenta un 1 más. En total, entonces, fueron 3 unos, que es el dato correcto.

Para obtener cada vez el número a comparar se utiliza la operación shift left (<<) del valor 1 el número de veces que sea necesario.

El algoritmo para esta rutina es:

1. iniciar número = 1 (el valor del término está implícito como dato de la clase term)
2. ¿número es menor o igual al valor del término?

   si -> 3

   no -> 7
3. ¿número AND valor = número?

   si -> 4

   no -> 5
4. aumentar en 1 el contador de 1's

5. hacer la operación shift left con número (una posición hacia la izquierda)
6. regresar a 2
7. terminar

La rutina unos( ) es una de las más importantes respecto del tiempo de proceso del programa. Esto es así, ya que esta rutina es llamada muchas veces durante los distintos cálculos.

2. void term::representados(int,int). La rutina representados( ) también es una función de la clase term. Se utiliza para obtener los valores binarios descritos por un término dado. Por ejemplo, el término {4,8,1} ó x100 simboliza los valores 4 ó 12. Esta rutina es importante en el momento de calcular un subconjunto de primeros implicados ya que permite establecer si distintos términos equivalen a los mismos valores ó minterms iniciales.

B. Rutinas varias

A continuación se presentan las rutinas del programa que no pertenecen a una clase específica. Estas rutinas pueden ser llamadas individualmente sin estar ligadas a términos específicos.

1. void ordena(term,int). La rutina ordena() se utiliza para ordenar una lista o vector de términos según el número de unos que posea cada término. Naturalmente utiliza la función unos() descrita anteriormente. El orden se realiza de menor a mayor y se opera únicamente al principio del programa, como en el método de McClusky.

2. int bits dif(term,term). Con esta rutina se puede determinar el número de 1's diferentes entre dos términos. Esto es importante, ya que la comparación de términos del método de McClusky es necesario que se realice entre dos términos que difieran en un solo bit. Esta rutina utiliza la instrucción XOR de C++. Como ejemplo comparemos los términos 1001 y 1100:

1001 XOR 1100 = 0101

A este resultado se le aplica la función unos() para contar el número de 1's que es 2. O sea, que los términos comparados difieren en 2 bits.

3. void condensa(term,term,int,int,int,int). La rutina condensa() es una de las más importantes del programa de diseño digital. Su función es condensar un vector de términos a partir de un nivel anterior, según el método de McClusky. Además, es en esta rutina donde se van apartando los primeros implicados. Como parámetros se pasan: el vector

de minterms, el vector donde estarán los términos ya condensados, el vector de marcas para los primeros implicados y los tamaños de cada uno de estos vectores.

El diagrama de flujo para esta rutina es el que se muestra en la figura 1.

En el diagrama:

elem ... puntero del primer vector que indica el término a ser comparado con los demás términos.

pasos ... puntero del vector de términos que indica el otro término a ser comparado cada vez.

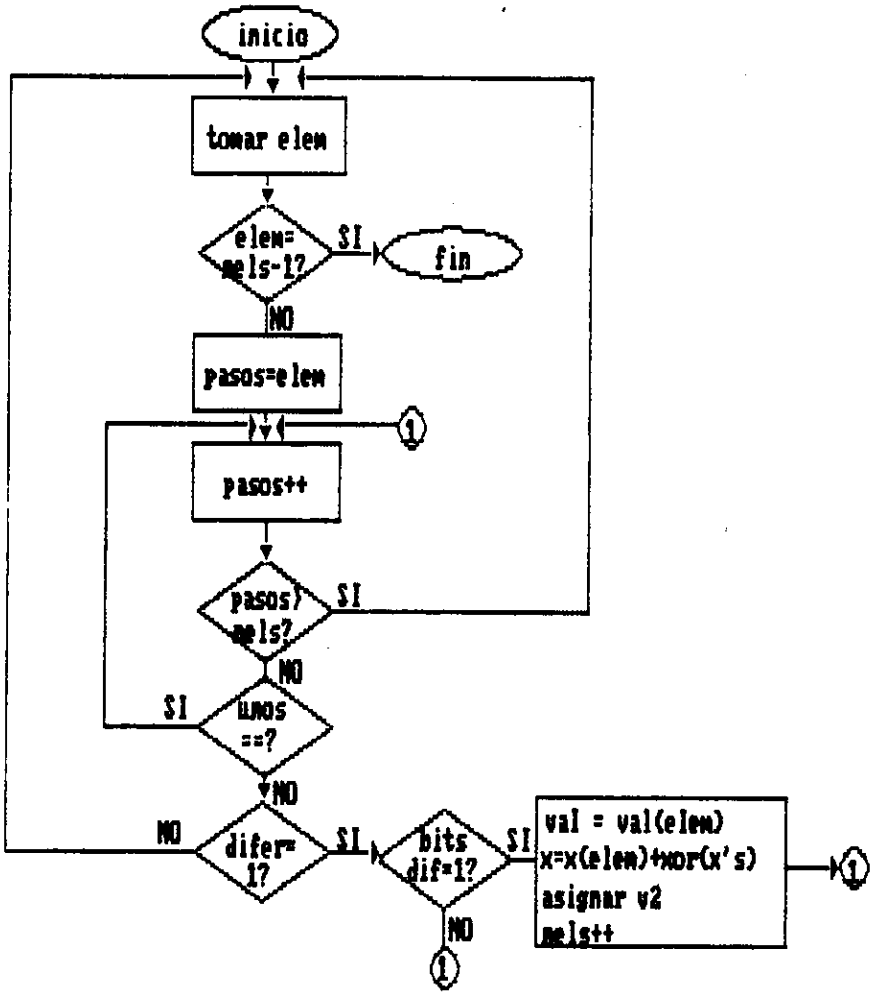
nels ... número de elementos del vector

Para condensar un par de términos en uno solo, la rutina toma en consideración lo siguiente:

- que los términos comparados no tengan el mismo número de 1's
- que el número de 1's difiera solamente en uno
- que los términos tengan un solo bit diferente
- que el don't care de ambos términos sea igual
- que ambos términos pertenezcan a la misma función de salida

Es importante notar que la rutina condensa() prácticamente realiza el proceso de minimización del método de

Figura 1
Diagrama de Flujo de la
rutina condensa( )



McClusky con algunas variantes. Entre estas variantes está el hecho de que la rutina va apartando de una vez los primeros implicados. En realidad solamente marca los términos que van siendo condensados. Los términos no marcados son entonces los primeros implicados. Además la rutina toma en

consideración el diseño para múltiples salidas, distinguiendo cada término por el número de función de salida a la que pertenece.

4. void elimrep(term,int,term,int). Al condensar un vector de términos con la rutina condensa(), el vector resultante puede tener términos repetidos. La rutina elimrep() se utiliza para eliminar estos términos repetidos para que no sean tomados en cuenta en el siguiente paso de minimización.

5. void sacaprim(int,int,term,int,term,int). Como en la rutina condensa(), solamente se marcan los términos que han sido condensados, es necesario, a partir de este vector de marcas, crear un vector de primeros implicados. La rutina sacaprim() se utiliza para ello. Como parámetros se le pasan: el vector de marcas de términos condensados, el vector de minterms y un vector para almacenar los primeros implicados en forma de términos.

6. void primeros_implicados(term,int,term,int). En la rutina primeros_implicados() están reunidas prácticamente todas las rutinas del programa de diseño lógico. El objetivo de esta rutina es obtener los primeros implicados a partir de un vector de minterms inicial. O sea que esta rutina es

la implementación directa del método de McClusky. Los pasos que sigue esta rutina son:

- ordena el vector inicial de minterms con la rutina ordena()
- condensa a un siguiente nivel con la rutina condensa()
- aparta los primeros implicados de ese nivel con la rutina sacaprim()
- elimina términos repetidos con la rutina elimrep()

Esta secuencia la realiza hasta que la rutina condensa() indique que ya no es posible condensar a un siguiente nivel.

7. void quita_repre(term,term,int). La rutina quita_repre() se utiliza para quitar, de un vector de minterms, los representados por un primer implicado dado. Esta rutina se utiliza en la parte de minimización, en la cual se obtiene un subconjunto de primeros implicados.

8. void SubPrimImp(term,int,term,int,term,int). SubPrimImp() se utiliza para sacar un subconjunto de primeros implicados a partir de un conjunto de primeros implicados básico. Esta rutina lo hace de forma aleatoria. O sea, va tomando, aleatoriamente, un primer implicado y eliminando del vector de minterms los representados por él. De esta

forma se logra eliminar, del conjunto básico de primeros implicados, aquéllos términos que representen a los mismos minterms iniciales.

La rutina SubPrimImp( ) no necesariamente proporciona un subconjunto mínimo de primeros implicados. Es necesario llamar a la rutina varias veces para obtener distintos subconjuntos entre los cuales se pueda escoger un posible subconjunto mínimo. Esta rutina es la que prácticamente realiza el proceso de minimización del método de McClusky.

VII. CONCLUSIONES

A. El proceso de automatización en el diseño de circuitos lógicos debe llevarse a cabo en forma parcial. Esto quiere decir que un programa de diseño lógico debe permitir la intervención del usuario y no automatizar el proceso completo, ya que la obtención de un diseño de circuito óptimo depende en gran medida de los criterios aplicados por el diseñador.

B. En un programa para diseño lógico es necesario tomar en cuenta las características específicas del lenguaje de programación. Es importante que este lenguaje permita la representación no numérica de la información, ya que los algoritmos de conmutación involucran el manejo de patrones de bits cuya operación no tiene significado numérico.

C. El programa de diseño lógico presentado en este trabajo cumple con las características básicas de un programa de su naturaleza. Sin embargo es necesario hacer notar que el programa presenta únicamente la base para la automatización del diseño digital. Para enriquecer el contenido del programa sería necesario tomar en cuenta, no sólo el criterio de número de compuertas, sino también otros criterios como fan-in, fan-out, etc.

VIII. BIBLIOGRAFÍA

Dietmeyer, D. 1978 Logic design of digital systems. 2nd. ed. Boston, Allyn and Bacon, Inc. 851 pp.

Lewin, D. 1974 Logical design of switching circuits. 2nd. ed. Great Britain, The English Language Book Society and Nelson. 404 pp.

Mano, M. 1982 Lógica digital y diseño de computadores. México, Prentice-Hall Hispanoamericana, S.A.. 636 pp.

Taub, H. 1983 Circuitos digitales y microprocesadores. México, Libros McGraw-Hill de México, S.A. de C.V.. 549 pp.

Turbo C++ getting started. 1990 U.S.A., Borland International. 268 pp.