ING. LUIS G. URIBE C.

Trabajo de Ascenso, Universidad Católica Andrés Bello

UBM
Máquina Booleana Universal

ING. LUIS GUILLERMO URIBE CATAÑO

UBM: Universal Boolean Machine

Universidad Católica Andrés Bello
Caracas, Venezuela

Contenido

PRÓLOGO

UBM: Algoritmo Universal para la Solución de Ecuaciones Booleanas, Combinacionales y Secuenciales en Microprocesadores.

Abstract

SE expone una metodología para resolver ecuaciones booleanas en tiempo real, con el énfasis puesto en los microprocesadores. Se presentan un programa, una forma de codificar las ecuaciones, y módulos auxiliares que sirven de soporte para convertir las ecuaciones formuladas de manera convencional al formato requerido en el microcomputador.

Introducción

Con el advenimiento de dispositivos de lógica programada, los llamados "***micros***" (**microprocesadores** y **microcontroladores**) se colocaron en el centro fundamental de los sistemas de control de toda clase y tamaño. Los programamos, entre otras cosas, para leer con regularidad un conjunto de variables externas que provienen del proceso que ha de controlarse, y para tomar decisiones en función de los valores que dichas entradas asuman. Las señales en cuestión pueden ser discretas o analógicas; en este último caso, deben convertirse a digitales.

La usanza dicta dos maneras primordiales de encarar el problema: O se elaboran programas diferentes y específicos para resolver cada circunstancia y éstos forman parte indisoluble del equipo que ha de instalarse ("firmware"), o se tiene un "interpretador" genérico dentro de tales dispositivos, a los cuales se les inyecta caso por caso un código que, al ser interpretado por el micro, resuelve la problemática en cuestión. El software introducido proviene de una estación de trabajo (PC) sobre la que se realiza la programación; ésta produce luego ese "módulo" que el micro

interpreta. En el primer caso se desarrolla en algún lenguaje seleccionado "ad-hoc" (C, Basic, Assembler) y cada situación requiere una codificación particular; en el segundo, suelen emplearse lenguajes orientados a la resolución de problemas de Control de Procesos como el "Relay Ladder Logic Programming", luego, el PC traduce esa programación a la que ha de ser interpretada por el micro y se la transfiere a éste mediante algún vínculo de comunicación (puerto serial, bus USB, conexión de red).

El primer método es dispendioso porque requiere elaborar un programa diferente para cada oportunidad, pero es un procedimiento "abierto" pues el ingeniero escoge sin ataduras el micro de su predilección. La segunda aproximación (interpretador genérico) es de más alto nivel, pero resulta un tanto "cerrada" desde que el soporte de programación es específico para cada dispositivo o PLC ("Programmable Logic Controller"); es decir, el diseñador se amarra con un lenguaje, un ambiente de desarrollo y un equipo final (micro) particular; y los de unas marcas no son intercambiables con los de otras [1].

En algunas circunstancias donde es fundamental la libertad de poder seleccionar, o de fabricar de manera autóctona algún equipo basado en micros –y que por tanto no poseerá ambiente de desarrollo propio de alto nivel, tipo PLC–, el ingeniero suele optar entonces por programar sus propios equipos, caso por caso como ya se dijo, con el resultante inconveniente de incurrir en costos recurrentes. Si él pudiera disponer de una herramienta que se programara una sola vez, pero que no dependiera de los costosos y particulares ambientes de desarrollo de alto nivel, podría producir sus dispositivos con los ahorros correspondientes, lo que elevaría su competitividad.

El autor tuvo la oportunidad de elaborar una de estas herramientas: La ***"Máquina Booleana Universal"***, bautizada con el acrónimo de ***UBM: "Universal Boolean Machine"***. ***UBM*** es un procedimiento universal, único y simple, capaz de resolver cualquier sistema de ecuaciones lógicas. El propósito de ***UBM*** es el de poner a la disposición de todo computador, pero de manera específica de los micros, un programa universal que evalúe cualquier ecuación lógica, combinacional o secuencial. Además, se propone una forma sencilla, eficiente y eficaz, de expresar las ecuaciones booleanas que relacionan las entradas con las salidas de cada problema particular (ecuaciones de entrada y salida). ***UBM*** está en capacidad de resolver esas ecuaciones para cada conjunto de entradas, y de producir automáticamente los valores de salida resultantes. Es de vital importancia resaltar que **no** hay que hacer todas las veces un programa particular que materialice los resultados que han de solucionar un problema determinado: ***UBM*** resuelve **todos** los casos posibles, dentro de las restricciones impuestas por las capacidades físicas del procesador (cantidad de memoria, número de puertos de entrada y salida, velocidad del dispositivo, etc.)

[1] En la actualidad hay algunos ofrecimientos de "ladder languages" gratuitos, escritos en C; cfr.: http://www.sourceforge.net/projects/classicladder ; en todo caso necesitan micros poderosos y costosos.

UBM se centra, con exclusividad, en aquellas partes o subsistemas de control en donde la información que se procesa es de estricta naturaleza discreta y que pueden modelarse como **Sistemas Digitales** convencionales, aprovechando el gran respaldo que el Álgebra de Bool ofrece en esta materia y, en particular, la enorme facilidad de poder expresar mediante simples ecuaciones lógicas, las funciones que relacionan entradas y salidas digitales tanto para problemas combinacionales como secuenciales.

Así, pues, aunque estas situaciones se resuelven de manera cotidiana en sistemas grandes basados en PCs, nuestro énfasis es el de proporcionar un procedimiento tan sencillo que pueda implementarse sin dificultad en microcontroladores de reducidos costos y prestaciones.

Reiteramos que en general, aquellas secciones de los sistemas de control que manipulan esta clase de problemas, expresables conforme a la metodología clásica de Sistemas Digitales, suelen hacerlo mediante la ejecución de *programas* específicos y diferentes que materializan en el micro algoritmos que *adquieren* valores digitales de entrada y en base a ellos *producen* resultados digitales en sus salidas. Cada problema que va a resolverse requiere la conceptualización de un algoritmo particular y la realización de programas distintos para cada ocasión. La propuesta de ***UBM*** consiste, empero, en una metodología basada en un procedimiento Universal, único y simple, capaz de resolver cualquier sistema de ecuaciones lógicas y con énfasis en microprocesadores. Además, la idea, como ya se ha dicho, es la de suministrarle al micro dentro de su memoria, junto con el programa universal único, una *representación interna* adecuada y conveniente de las ecuaciones booleanas que relacionan las entradas con las salidas en cada problema particular.

El levantamiento y formulación de tales funciones sí es una actividad específica e ineludible que ha de efectuarse para resolver los diversos problemas; forma parte del indispensable proceso de análisis que debe realizarse caso por caso, y es labor común del diseño, tanto si se trata de la aproximación clásica como si se emplea la metodología que aquí proponemos. Pero a partir de allí, nuestro único y universal procedimiento, ***UBM***, aplicará un algoritmo que arrojará resultados ***sin que haya que codificar ningún programa diferente para cada nuevo conjunto de ecuaciones***. Esta es la separación fundamental entre la metodología convencional y la de ***UBM***, a nivel de los microcontroladores.

Plan del Documento

Nuestra presentación se encuentra distribuida a lo largo de siete (7) capítulos. En el primero, *Metodologías Convencionales*, se pretende ubicar el tema central mostrando de forma somera las técnicas comunes de programación que se emplean en la actualidad para la resolución de la clase de situaciones que nos atañe, y el entorno más general en el que estos problemas se desarrollan. Se ofrece un resumen de las técnicas aplicables a micros, a fin de facilitar la comparación con el procedimiento aquí propuesto. El material que se recoge en esta sección es breve por necesidad pues una presentación completa coparía varios volúmenes. Se realiza una reseña somera sobre métodos convencionales de empleo común en la práctica profesional, como la programación de PLCs: "Relay Ladder Logic Programming"; técnicas de minimización y solución de ecuaciones booleanas en programas al estilo de "Espresso" y "Spice" (PSpice y su heredero, el "Electronic Workbench"). Se analizan las diversas formas que pueden emplearse en los micros para codificar tanto las ecuaciones lógicas como la información de entrada y salida.

En el segundo capítulo, *Solución de Ecuaciones Booleanas*, se describe la forma convencional de resolver ecuaciones expresadas tanto de manera canónica como en forma normal, simplificadas.

El tercero, ***XBIN***, *o Codificación Binaria Extendida*, explica nuestra selección para la representación interna de las ecuaciones booleanas; ***XBIN*** constituye el corazón de ***UBM***. El cuarto capítulo, ***UBM:*** *Máquina Booleana Universal,* corresponde al tema central. Allí se realiza paso a paso la presentación del método, y se muestra con la mayor sencillez posible su aplicación a circuitos combinacionales. En el capítulo 5, ***UBM*** *en Sistemas Secuenciales,* se amplía el algoritmo para incluir la variable tiempo.

En el sexto, *Herramientas de Soporte,* se introduce un conjunto de programas utilitarios que, si bien no son necesarios para la aplicación de ***UBM***, simplifican de manera considerable el trabajo de adaptar la formulación convencional de las ecuaciones booleanas a la representación interna requerida. El capítulo 7 compendia el proceso y en él se desarrollan varios *Ejemplos* para ilustrar con detalle el procedimiento a fin de despejar dudas. Por último se incluyen en la bibliografía referencias a material selecto que permitirán al lector interesado estudiar en profundidad algunos de los temas tratados, en especial el de la programación de los PLCs.

Si el lector sólo pudiera dedicar un mínimo de atención a este documento, los capítulos esenciales son: el tres, el cuatro y el cinco. Las herramientas, en el 6, vienen bien, y los demás son accesorios.

Resulta un tanto curioso por decir lo menos, que técnicas tan sencillas como eficaces al estilo de ***UBM*** no se empleen de manera rutinaria en la docencia, en asignaturas

orientadas al estudio de los microprocesadores y microcontroladores, y muchas veces tampoco se apliquen en la práctica profesional, pero de hecho es corriente ver que los problemas de la naturaleza a la que aquí nos referimos se programen uno a uno cada vez, desperdiciando sin necesidad esfuerzos apreciables. Es de esperar, por tanto, que este documento sea de utilidad como referencia a los interesados en simplificar sus vidas, que buena falta nos hace, y de complemento a la literatura cotidiana en la materia.

UBM ha sido aplicado con éxito en proyectos de naturaleza industrial, con los resultados previstos.

El autor, que ha tenido la fortuna de haber transitado durante 36 años el camino del desarrollo tecnológico, recorriendo desde el relé y el tubo de vacío hasta el Pentium y el Internet, ofrece con el mayor placer esta pequeña contribución.

Ing. Luis G. Uribe C.

Caracas, Noviembre de 2005

Capítulo 1

METODOLOGÍAS CONVENCIONALES

Cómo se trabaja en la actualidad...

Perspectiva

LOS dispositivos modernos de lógica programada (microprocesadores y microcontroladores) han logrado un uso preponderante en su aplicación como elementos de automatización, incorporados dentro de una amplísima variedad de sistemas que cubren desde los más sencillos ("embedded processors") al estilo de los hornos de microondas, hasta los más complejos instrumentos como los medidores de parámetros eléctricos o los controladores programables PLC, a través de toda una gama de sectores como el automotriz, el industrial, el laboratorio, consultorios médicos, el hogar, la oficina...

Cuando se aplican tales componentes al Control de Procesos, la metodología general de trabajo suele ser como sigue: Un lazo sin fin dentro del micro le transfiere el mando con periodicidad (cada 10 milisegundos [2]) a la Rutina de Control de Procesos RCP. Allí se procede a leer los valores externos que se analizan para determinar si, por ejemplo, algunos contactos están abiertos o cerrados, o si algunas mediciones de variables continuas han sobrepasado o no determinados límites. El programa decide qué salidas deben generarse en base al estado y valores de la información leída.

Dividiremos en dos el asunto que nos concierne: El procesamiento de las variables analógicas y el de las digitales. Nuestra metodología no cubre el tratamiento de variables analógicas; se especializa sólo en la manipulación de los sistemas digitales.

---

[2] 10 milisegundos es casi la norma

Muchos elementos de control industrial usan como entradas, simples interruptores externos, y producen como salidas, aperturas o cierres de contactos: Son problemas 100% digitales. A éstos se aplica a la perfección nuestra propuesta, y también a aquellos que puedan tomar decisiones basadas en lecturas analógicas, siempre que las alternativas se tomen sobre resultados booleanos obtenidos de la manipulación de esos valores no digitales; por ejemplo, el hecho de que una temperatura sea mayor que una referencia, o no lo sea (la variable y el parámetro de comparación son analógicos en su origen, pero el valor de verdad de la comparación entre ellos es booleano).

Si alguna parte del procesamiento fuera en esencia no digital, se la deberá tratar por separado y emplear nuestra metodología en la sección digital correspondiente, a la que sí aplica.

En muchas ocasiones ocurre que la modelación de los fenómenos que van a controlarse puede expresarse de la manera más natural mediante Ecuaciones Lógicas. Observemos, como ilustración, un caso tomado de la vida real:

Descripción de un problema de ejemplo:

Desarrollar un sistema con un microprocesador para controlar 2 cargas que consumen electricidad en un restaurante:

- *Tres hornos para la preparación de panes, tortas y similares y*
- *El sistema de aire acondicionado.*

El panadero podrá encender los hornos a discreción, pero sólo fuera del horario comprendido entre las 11 a.m. y las 2 p.m. El aire acondicionado se apagará si la demanda eléctrica del establecimiento sube por encima de 160 KVA, y no volverá a encender antes de 10 minutos de haberse apagado, para proteger los compresores de sucesivas conexiones y desconexiones.

Hay dos señales de ***salida****, que son habilitadoras ("permisivas") para:*

- **R0**: Los 3 hornos
- **R1**: Aire acondicionado

Las variables de ***entrada*** *son:*

- ***H****: Horario para habilitar los hornos (0: entre las 11 a.m. y las 2 p.m.; 1: en el resto)*
- ***D****: Demanda (1: está siendo sobrepasada)*
- ***T****: Temporizador de 10m para el AC. (1 = expiró: pasaron al menos 10 minutos desde su activación)*

Una señal auxiliar de salida dispara internamente el temporizador ***TT****: Trigger Timer, después de lo cual éste no depende de su entrada hasta terminarse el intervalo establecido de 10 minutos.*

Obsérvese que a pesar de la presencia de diversas variables analógicas (el horario, el valor de la demanda y varios temporizadores), el problema es del estricto ámbito de los sistemas digitales, y su formulación en términos booleanos es sencilla y resulta muy conveniente.

Técnicas de Uso Común

Por lo general hay tres métodos alternativos que se emplean para resolver este tipo de problemas. La **Aproximación Convencional** consiste en formular la situación usando expresiones lógicas, mediante el álgebra de Bool, y en confeccionar luego un programa, diferente y específico para cada conjunto de ecuaciones, en un lenguaje de alto nivel como podría ser algún dialecto del Basic, el "C" o cualquier otro disponible en el dispositivo de control para esos menesteres. El segundo método se basa en formular la solución con un **Diagrama de Escalera** ("Relay Ladder Logic Programming"), que se usa para programar equipos de la estatura de los PLC convencionales u otros que contengan entre su repertorio un interpretador apropiado. El tercero emplea **técnicas "ad-hoc"**, propias de los diferentes manufacturantes de equipos.

Aproximación Convencional

Como nuestra propuesta va dirigida más hacia los pequeños procesadores que por razones de costo y espacio no pueden incluir tales interpretadores; veremos que en este caso lo usual en la actualidad es realizar, para los micros, programas o rutinas con un aspecto similar al del siguiente ejemplo:

```
/* BOOL solve_f1( BYTE f1 ): Rutina para resolver la siguiente
 * ecuación booleana, expresada como la suma de dos términos
 * de producto, en forma canónica:
 *    f1 = A  * B' * C' * D' * E * F * G' * H  +
 *         A' * B  * C  * D' * E * F * G' * H'
 * Se ha decidido representar cada una de las 8 variables,
 * de la A a la H, por el correspondiente bit tomado de
 * izquierda a derecha, de los 8 que conforman el parámetro
 * 'f1' de la función solve_f1().
 *
 * solve_f1() retorna 0 o 1 lógico, según sea la evaluación.
 *
```

```
 *   Cfr. Digital Design with Standard MSI & LSI;
 *   Tomas R. Blakeslee, Wiley, 2e, 1979,
 *   pp 169 y ss.
 *
 */

typedef enum { FALSE, TRUE } BOOL;
#define  F11 0x8D  // Hex para 1000 1101 (A*B'*C'*D' * E*F*G'*H)
#define  F12 0x6C  // Hex para 0110 1100 (A'*B*C*D'  * E*F*G'*H')

BOOL solve_f1( BYTE f1 )
{
   if( f1 == F11  ||  f1 == F12 )
       return TRUE;
   else
       return FALSE;
}
```

En el ejemplo se ha representado cada variable por un bit, seleccionado en forma arbitraria los 8 bits de izquierda a derecha, comenzando con la variable A y terminando con la H. Como la forma es canónica, todas las variables aparecen una y sólo una vez en cada término de producto, afirmadas o negadas, y la representación más simple es, entonces, colocar un uno para la variable afirmada y un cero si está negada. A cada uno de los términos de producto se le asocia una constante que equivale a la combinación canónica de ceros y unos que harían que su valor fuera "verdadero" (`F11` y `F12`). La evaluación que realiza la función `solve_f1()` consiste en ver si el valor que se le pasa como parámetro –y que está coformado por los ceros y unos de todas las variables, leídos al momento de la evaluación–, corresponde al valor definido para alguna de las dos constantes (una por cada término de suma); de ser así, la función retorna el valor "verdadero"; de lo contrario el resultado es "falso". De esta manera, y a modo de ejemplo, si en el momento en que se leen las variables éstas valieran todas cero (`f1 = 0x00`) o todas uno (`f1 = 0xFF`), la función no sería igual a ninguno de los dos términos canónicos de producto, `F11` o `F12` y el resultado de la evaluación sería "falso". Si las variables conformaran el valor "`1000 1101`", el valor leído f1 resultaría, al hacer la comparación, igual a la constante `F11`, lo que querría decir que el resultado de la evaluación de la función sería "verdadero".

Que quede claro que sería necesario escribir una rutina similar para cada ecuación lógica que debiera resolverse: `solve_f1, solve_f2 ... solve_fm`, que tendrían más o menos términos canónicos de producto, según fuera el caso, y habría que definir

las constantes con las cuales comparar dicho términos de la función expresada como Suma de Productos: **F11, F12...F1n; F21, F22...F2j; Fk1, Fk2 ... Fkx.**

El lazo sin fin del que se habló antes puede parecerse al del siguiente código, que le transfiere con regularidad el mando a las Rutinas de Control de Procesos, Analógicos y Digitales. Allí se procede a leer los valores externos que se procesan para determinar si algunos contactos están abiertos o cerrados, o si algunas mediciones de variables continuas han sobrepasado o no ciertos y determinados límites. El programa decide qué salidas deben generarse en base al estado y los valores de la información leída, tanto la externa (interruptores) como la interna (relojes, temporizadores, contadores, comparadores).

```
/* Parte del lazo sin fin que transfiere con regularidad el
 * mando a la RCP. Se muestra la parte correspondiente a la
 * lógica digital RCP_digital(). La analógica no se trata.
 * El sistema suele incluir Relojes, Temporizadores y Contadores.
*/
void RCP_digital ( void )
{
   getInput( f1, f2, ..., fn);
   getClock( c1, c2, ..., ci);
   solve_f1( f1 );
   solve_f2( f2 );
   ...
   solve_fm( fm );

   updateOutput(o1, o2, ..., oj);
}
```

```
...
   while( 1 ) {    // Lazo principal: for ever
   ...
       RCP_analogica();
       RCP_digital();
       sleep( TIME );  // normalmente 10 milisegundos
   ...
   }  // end while
...
```

Este procedimiento **no** es universal; es particular para cada conjunto de ecuaciones.

Blakeslee, el autor arriba citado al elaborar la función "solve_f1", incluye en su libro el siguiente comentario que ilustra con toda precisión tan común forma de trabajar:

"*We normally never use logic equations in program design. The* ***functions are defined*** *naturally* ***by making flow charts*** *of the desired functions,* ***then writing*** *a sequence of* ***instructions to implement the flow chart.***" (Blakeslee, op. cit., pp 171; el subrayado y el resaltado son nuestros).

Es evidente que cada programa así, requiere un diagrama de flujo particular, lo que resulta en rutinas específicas para cada oportunidad. Esa es la usanza, y es lo que con precisión *UBM* se propone evitar.

"Relay Ladder Logic Programming" en la Estación de Trabajo

La segunda metodología surgió a partir del desarrollo del electromagneto, cuando los ingenieros de Control de Procesos automatizaron las líneas de ensamblaje de las fábricas empleando lógica fundamentada en relés (con bobinas activadas tanto en AC como en DC).

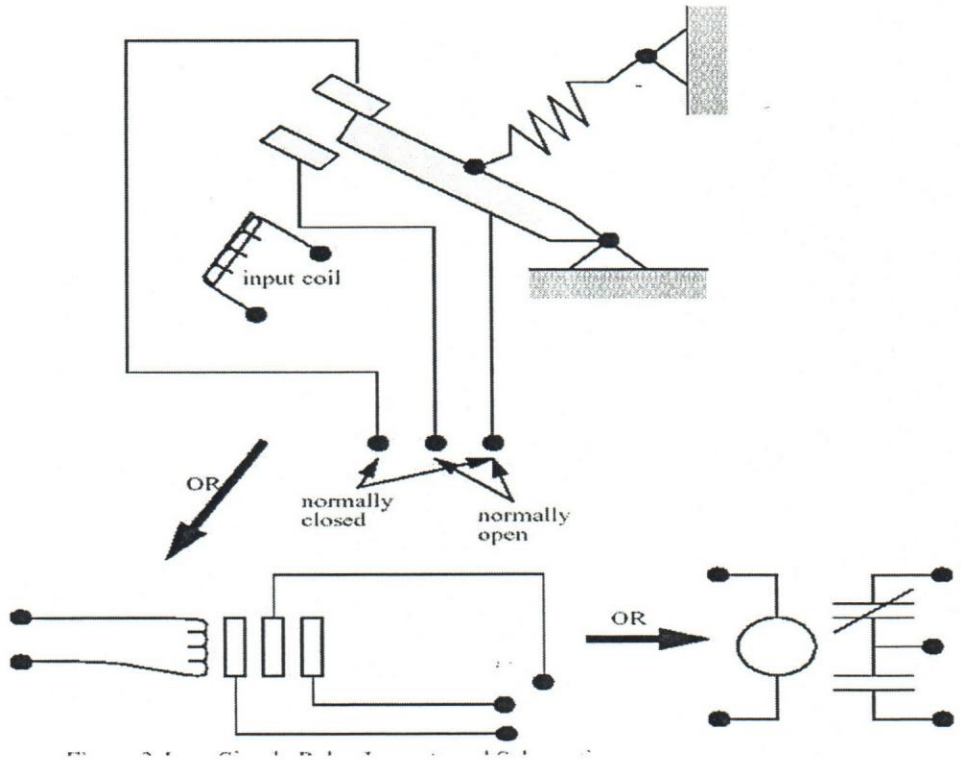


Figura 1.1
Tres formas de ver un relé: Esquema electromecánico, diagrama eléctrico y diagrama lógico. Los voltajes que manejan las bobinas a veces son los mismos que se conectan a los contactos y a veces no. Por ejemplo, pueden ser 5VDC en la bobina y en los contactos, o 5VDC en las bobinas y 120VAC en los contactos, o cualquier combinación.

La programación para obtener una u otra función dentro de una planta industrial se facilitaba al emplear ciertos tableros metálicos en los cuales se alojaban los relevos, y que ofrecían acceso sencillo tanto a las bobinas como a sus contactos, mediante borneras o tomas especiales. Solía haber un sistema de rieles que alimentaba el conjunto (ver figuras 1.2 y 1.3), el "vivo" por una línea (L1), el neutro por la otra (L2), y en el medio se disponían las bobinas y los contactos. La apariencia del conjunto era como la de una "escalera", siendo sus largueros los dos extremos de la alimentación, con tantos peldaños (líneas horizontales) como circuitos de control fueran a

implementarse. Mediante cables se interconectaban las bobinas y los contactos de acuerdo a la operación que se deseaba automatizar.

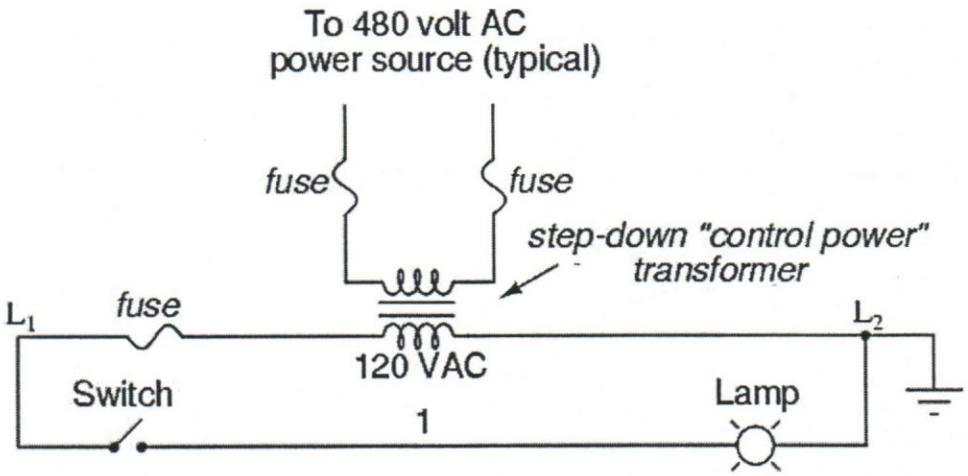


Figura 1.2
Estructura de Escalera, para alimentación AC

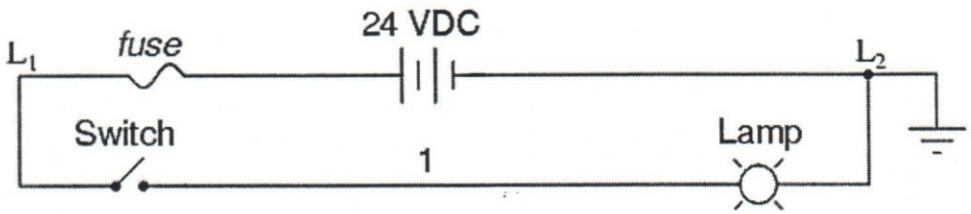


Figura 1.3
Estructura de Escalera, para alimentación DC

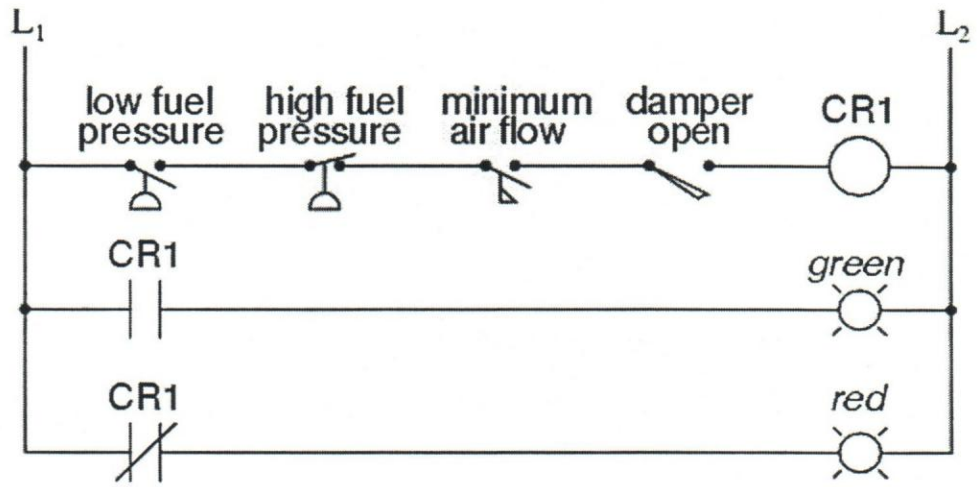


Figura 1.4
Estructura de Escalera; como ejemplo se muestran algunos interruptores de una planta, y los contactos normalmente abiertos (NO) y normalmente cerrados (NC) del Relé de Control CR1 ("Permissive and interlock circuit")

La Fig. 1.4 muestra como ejemplo un "Permissive and interlock circuit" de una planta.

A mediado de los años 60 se introdujeron los PLC ("Programmable Logic Controllers") con electrónica transistorizada capaz de suplir muchas de las funcionalidades de los antiguos tableros; se mantuvieron los relés (que pueden ser tanto dispositivos electromagnéticos reales, como semiconductores basados en tiristores), ya no para diseñar la lógica con ellos, sino como interfaz con el proceso final, a fin de manejar con comodidad las diversas cargas mediante sus contactos. Para ese entonces, numerosos ingenieros Mecánicos y de Control de Procesos habían sido entrenados en la automatización de plantas vía "Relay Ladder Logic Programming", por lo que se mantuvo el mismo paradigma en los PLCs, a fin de evitar costosos reentrenamientos. Cuando con posterioridad se incorporaron los micros a los controladores lógicos, se continuó manteniendo el mismo mecanismo de programación, sólo que ahora se empleaba una computadora personal, o estación de trabajo, y en ella, a través de una interfaz gráfica se recreaba en la pantalla el antiguo mecanismo de cableado, como si se estuviera frente al tablero original en la fábrica; al final, la estación de trabajo compilaba un programa, adecuado para el respectivo PLC, que al ejecutarse en aquél terminaba materializando, en la planta, las funciones indicadas por el ingeniero.

Así operan muchos de estos equipos en la actualidad.

Veamos un ejemplo para ilustrar un poco más ésta forma de programación con lógica de escalera. Supongamos que tenemos un motor trifásico reversible y queremos garantizar que podemos iniciar su movimiento en un sentido o en el otro, pero no en ambos al mismo tiempo. La Fig. 1-5 muestra los interruptores provenientes de dos (2) contactores (un contactor es un relé con capacidad de manejar cargas de grandes corrientes), conectados a los 3 devanados del motor de tal forma que se apliquen las tres fases de alimentación eléctrica en la secuencia A, B, C para el giro en un sentido, y B, A, C en el caso contrario. Si los dos contactores están desenergizados se detiene el movimiento del motor. Los contactores M1 y M2 no aparecen dibujados en la Fig. 1-5; sólo 3 de los contactos provenientes de cada uno de ellos (para un total de seis).

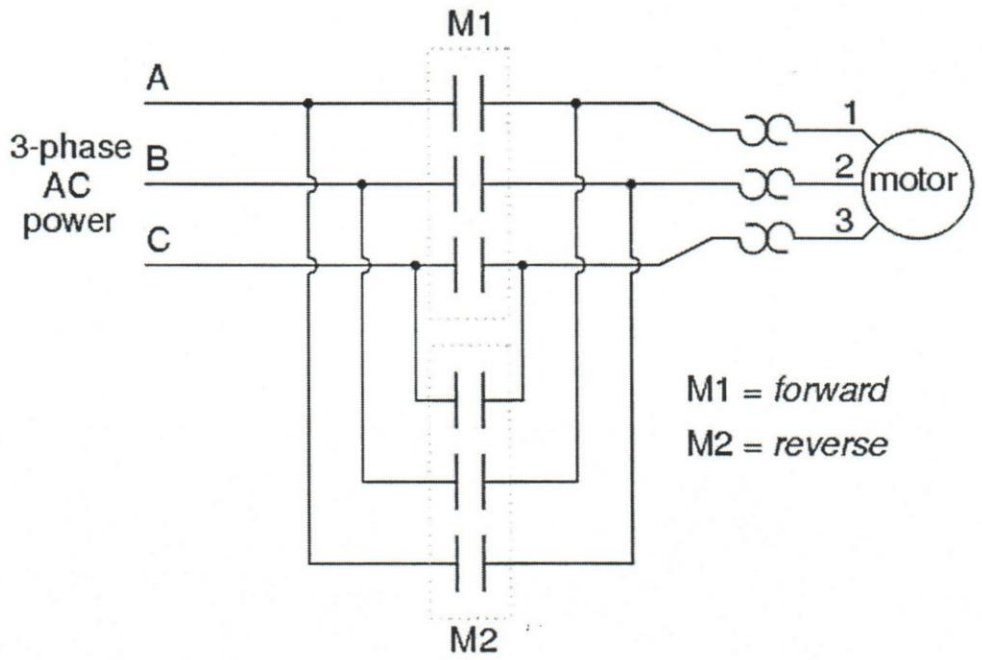


Fig. 1-4
Se aplican las fases A, B, C para giro en un sentido, y B, A, C en el otro, mediante los contactores M1 y M2

Una primera aproximación a la lógica que hay que establecer para controlar el giro del motor en un sentido o en el otro, o para hacer que permanezca detenido, se ve en la Fig. 1-6. Los número identifican los cables:

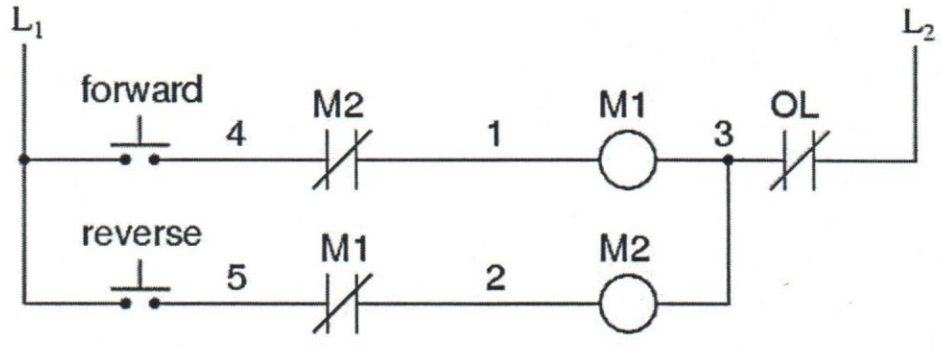


Fig. 1-6
Se garantiza que los giros en un sentido y en otro son mutuamente exclusivos

Los interruptores M1 y M2 de los respectivos contactores están alambrados de tal manera que garantizan que las órdenes para hacer girar el motor en un sentido o en el otro sean mutuamente exclusivas: Si se ha iniciado una operación "forward" se abre automáticamente la línea que podría producir una operación "reverse", y viceversa. Además de los contactores y sus interruptores asociados, se ven los pulsadores empleados para los comandos de giro (forward y reverse), y un (1) contacto permisivo que proviene de sensores térmicos de sobrecalentamiento (OL: Overload) instalados en cada devanado del motor. Note que éste se detiene cuando deja de oprimirse el mando correspondiente; por eso no se requiere un botón de parada, pero esta conexión obliga al operador a mantener oprimido durante toda la operación el pulsador escogido.

En la Fig. 1-7 puede verse un circuito de control un tanto más complejo, que añade un botón para detener el motor (stop) y que hace que el giro seleccionado se mantenga, aún después de retirar el respectivo pulsador (autoretención). Note que la maniobra para invertir el giro del motor es como sigue: Hay que oprimir primero el botón de parada y luego activar el pulsador inverso. El botón de parada **abre** su contacto cuando se lo oprime (contacto normalmente cerrado, NC), a diferencia de los otros dos, que **cierran** la conexión cuando los presionamos (contactos normalmente abiertos, NO).

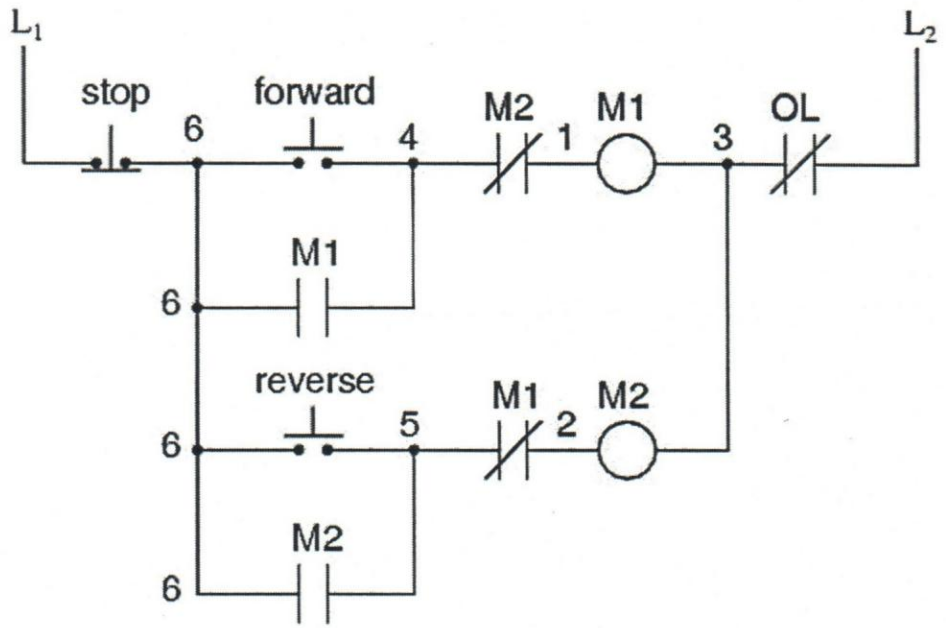


Fig. 1-7
Este diagrama garantiza que los giros en un sentido y en otro sean mutuamente exclusivos; se añade el control de parada, y ahora basta con oprimir y soltar (pulsar) los botones para ejecutar la operación asociada (autoretención).

Para terminar de ilustrar este problema y su programación mediante Diagramas de Escalera, la Fig. 1-8 presenta la inclusión de dos relés de retardo (TD, Time Delay 1 y 2) para garantizar que el operador se encuentre imposibilitado de invertir el sentido de giro del motor de forma prematura y forzar una espera mínima prudencial, calculada para proteger al motor, que podría haber continuado girando aún después de suprimírsele la energía con el botón de Stop, debido a la inercia de la carga acoplada. Aplicar alimentación inversa en este caso ocasionaría grandes corrientes y reduciría potencialmente la vida útil del motor. Observe que en el ejemplo, el retardo en los contactos se produce cuando éstos deben regresar a su estado normal (Off-Delay: Posición de cerrado en la figura). Para invertir el sentido de giro del motor la maniobra sería como sigue: Primero se oprimiría el botón de stop, lo que de inmediato desenergizaría el motor, pero hasta pasado el retardo correspondiente no se activaría el contacto normalmente cerrado del relé temporizador asociado; esto impediría al operador manipular el control inverso durante ese lapso, o al menos haría inútil su activación, en caso de que se manipulara el pulsador antes de tiempo.

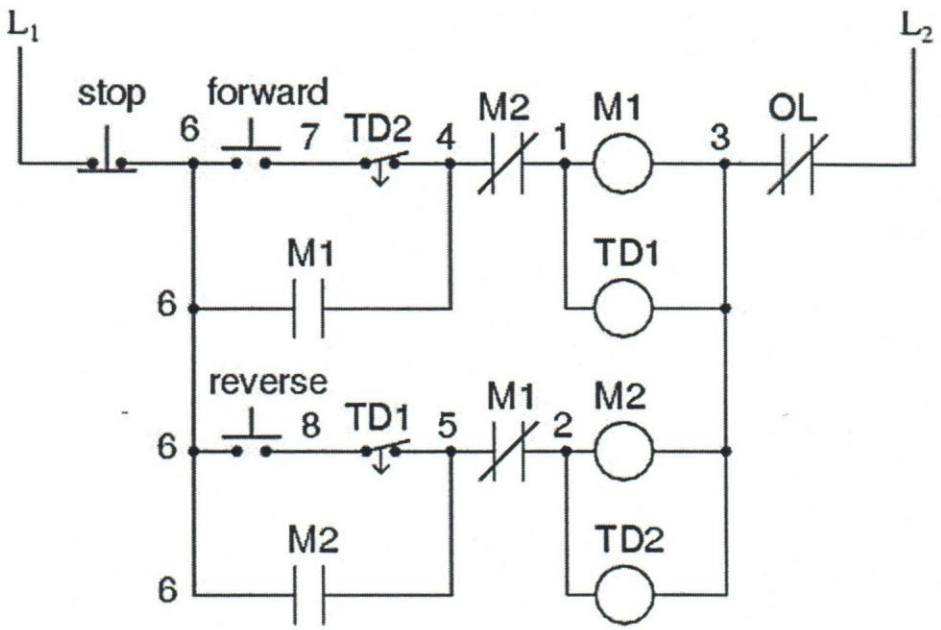


Fig. 1-8
Se añaden retardos para evitar cambios de giro
en momentos inconvenientes.

Por último, la Fig. 1-9 representa el mismo circuito de la 1-8, simplificado:

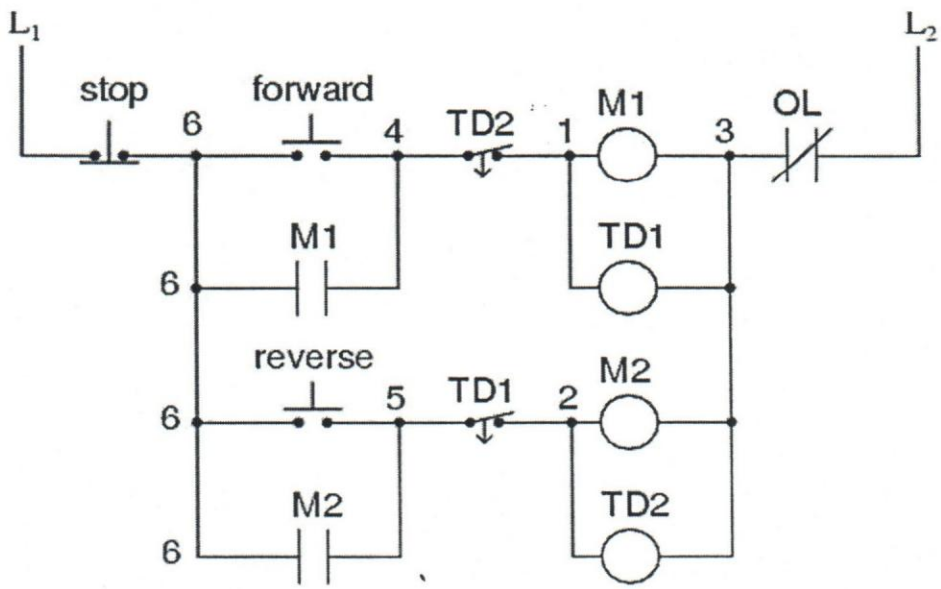


Fig. 1-9
Circuito final, simplificado

Puede resumirse lo dicho hasta aquí indicando que la programación moderna mediante "Relay Ladder Logic Programming" consiste en dibujar en una pantalla, circuitos de apariencia similar a los arriba mostrados, luego de lo cual puede simularse en la estación de trabajo el comportamiento del sistema, para verificar que el resultado sea satisfactorio, y se procede entonces a compilar el "Relay Ladder Logic Programming" y producir un código apropiado para el PLC, que éste debe ejecutar. Normalmente la estación de trabajo, o un Laptop, se conecta con el PLC mediante algún vínculo de comunicaciones apropiado (puerto serial, USB o interfaz de red) y se procede a alimentarle el código.

"Relay Ladder Logic Programming" en el PLC

En el **Anexo A** se muestra el funcionamiento ***desde el punto de vista del PLC;*** se incluye, como ejemplo, una sección del código residente en un PLC comercial basado en un PC industrial, con un microprocesador Intel de la familia 80x86.

Comentario Final Sobre Programación de Escalera

Podemos intuir que el "Relay Ladder Logic Programming" se justifica en procesos complejos, que requieran la aplicación de equipos costosos como los PLC, que incluyen el interpretador de los Diagramas de Escalera, pero es casi seguro que esa metodología deba quedar descartada en las aplicaciones más pequeñas realizadas con microcontroladores; lo mismo podría aplicar para negar la posibilidad de emplear aquí otros lenguajes como el **C**, y menos si tienen que estar residentes dentro de la de por sí pequeña aplicación, como sería el caso del **Basic** y demás interpretadores.

Otros Programas Importantes

Ya hemos visto la aproximación convencional para la resolución de ecuaciones lógicas en microprocesadores, y la que se emplea en los PLCs (programación de escalera). Hay un grupo de programas, como el "**Espresso**" y el "**Spice**" (ambos originales de UC Berkeley), que fueron predecesores de gran cantidad de software moderno, como el **PSpice** (Spice para PC, de OrCad) y el **Electronic Workbench**, heredero también del motor de Spice para la sección digital. Muchos tenían como objetivo la **minimización** de funciones; tal es el caso de Espresso, y luego vino la transición hacia la **simulación**, lo que introdujo la necesidad de **solucionar** las ecuaciones, además minimizarlas.

Dos aspectos importantes pueden analizarse en esos programas: La **representación interna** de las funciones booleanas, **y el método de resolución** de ecuaciones. Recordemos que hasta ahora las ecuaciones se han representado:

- Como expresiones booleanas, **canónicas**, en suma de productos, en las cuales los bits representan, por su posición, las diferentes variables, y como todas ellas aparecen en cada término (por ser expresiones canónicas), basta con indicar si valen 0 o 1,

- Como expresiones **normales** (no canónicas); en este caso, no todas las variables participan en todos los términos de producto, por lo que es necesario identificar cuáles aparecen y, si lo hacen, qué valores asumen.

"Espresso" emplea una representación interna de las fórmulas en suma de productos; si en algún término aparece la variable afirmada va como un "0"; si está negada se representa como un "1" (Cfr. "Espresso.5, input file format for espresso", y "Espresso.1, página del manual, Diciembre 28 de 1984"). En la representación externa (los archivos en los que el usuario define las ecuaciones) si no figura una variable se indica como un tercer símbolo: el "-".

Ejemplo de representación externa estándar en formato estándar de Berkeley para la descripción física de los PLA:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4                                  01 10     011
.o 3                                  01 11     100
.type fr                              10 00     010
.pair 2 (1 3) (2 4)                   10 01     011
.phase 011                            10 10     100
00 00     000                         10 11     101
00 01     001                         11 00     011
00 10     010                         11 01     100
00 11     011                         11 10     101
01 00     001                         11 11     110
01 01     010                         .end
```

UC Berkeley especificó un formato estándar de archivo para la definición de PLAs; a él se acogen tanto "Espresso" como "Spice" (y otros muchos productos). En la tabla anterior se habla del "On-set", o Suma de Productos; existe también la representación en Productos de Sumas: es el "Off-set".

Veremos pronto que para nuestras necesidades en el ambiente de los **micros**, son preferibles las representaciones **NO** canónicas porque, en general, son mucho más reducidas (se las emplea minimizadas) y esto es lo que conviene en ambientes de recursos mínimos, tanto de memoria como computacionales.

Puede observarse del análisis de programas como Spice, que emplean representaciones **canónicas**, por lo que basta con indicar si las variables participan de manera afirmada o negada. Eso se realiza, por lo común, colocando un 0 cuando la variable está afirmada, y un 1 cuando se encuentra negada. Así, los términos de productos se representan mediante posiciones de memoria (bytes si se trata de conjuntos de 8 o menos variables, enteros y enteros largos si son 16 o 32 variables), de las cuales se asignan, de forma arbitraria, una correspondencia entre las variables y sus bits (cfr. el antecedente: "Aproximación Convencional"; en particular, el ejemplo "solve_f1")

La representación en Forma Canónica, Suma de Productos (SP) es relativamente sencilla; sin embargo, también observamos en esos programas que mientras menos términos SP haya, más expedito resulta el trabajo del microprocesador. Esto nos da un indicio de lo conveniente que sería proveer las ecuaciones lo más simplificadas que fuera posible, lo cual ***excluye***, desde luego, el empleo de la forma Canónica.

Capítulo 2

SOLUCIÓN DE ECUACIONES BOOLEANAS

Modalidades convencionales...

SE presentan dos alternativas comunes para la solución de ecuaciones booleanas: la que recibe como insumo una formulación ***canónica***, y la que se emplea en los microcomputadores, basada en la representación ***normal***, simplificada, lo que hace que se economice en espacio y se logren mejores tiempos de ejecución, por lo que lucen más apropiadas para nuestros propósitos.

Solución de Ecuaciones Canónicas, SP

Supongamos que tenemos la siguiente función Canónica de A, B, C, escrita en términos de Suma de Productos [3]:

```
F(A, B, C) = ~A*~B*~C + ~A*B*~C
```

---

[3] Hemos empleado el símbolo "~" para representar el NOT o negado de la variable (o expresión, si está en paréntesis) inmediatamente a su derecha; es un operador unario (afecta sólo a una variable o expresión parentizada). En cuanto a los operadores binarios (que involucran dos elementos), para identificar el AND se ha escogido el "*" y para el OR se ha usado el más convencional "+". El símbolo "=" separa el lado izquierdo del derecho en la ecuación.

Recordemos que la designación de "Canónica" implica que figuren en todos los términos exactamente una de cada una de las variables independientes, bien sean afirmadas o negadas.

Un diagrama para bosquejar la solución de esta función sería el siguiente:

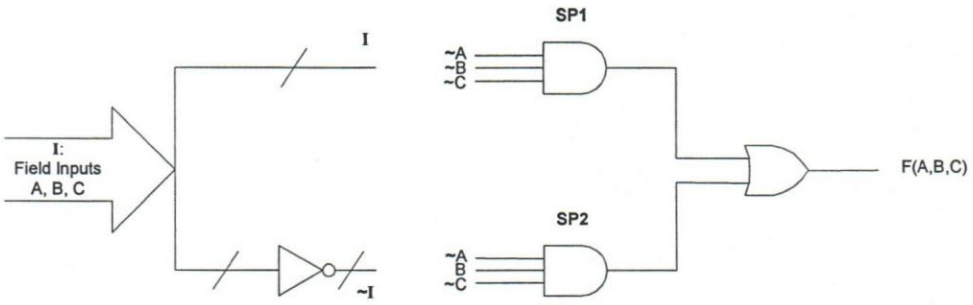


Fig. 2-1
Diagrama explicativo de la solución de ecuaciones canónicas.

A la izquierda de la Fig. 2-1 se representa el Vector de entrada **I**, formado en este caso por 3 bits que corresponden a las variables A, B y C. El sistema adquiere dicha información y la distribuye para ser empleada por el circuito, tanto de manera afirmada **I** (A, B, C) como negada **~I** (~A, ~B, ~C).

En nuestro ejemplo la ecuación tiene dos términos de Suma de Productos: SP1 y SP2. Si hubiera que resolver una ecuación de este tipo empleando un microprocesador deberíamos leer uno a uno los términos **SP** (SP1 y SP2) y para cada uno de ellos, dependiendo de cómo aparecen las variables, si afirmadas o negadas, aplicar el vector de entrada **I** (las variables...) y establecer en el micro la operación equivalente a cada AND.

Queda claro que las entradas son bits, valores digitales, pero, ¿cuál es la forma más conveniente para expresar y suministrarle los ***términos*** **SP** al microprocesador (la ecuación)? Ellos se formulan normalmente representando cada variable mediante caracteres alfabéticos (A, B, C en nuestro ejemplo) y podrían inclusive ser palabras u otras expresiones alfanuméricas que resultaran apropiadas para la descripción del problema, como, por ejemplo:

```
Resultado(Var1,Var2,Var2)= ~Var1 * ~Var2 * ~Var3 +
                           ~Var1 *  Var2 * ~Var3
```

Si se deseara suministrarle al microcomputador el conjunto de ecuaciones que caracteriza un problema, empleando directamente las definiciones alfanuméricas de las variables, se precisaría tener códigos que identificaran los diversos componentes: Las variables y las operaciones que entre ellas se establecen (NOT, AND, OR), para posteriormente producir un resultado. Aproximaciones de este tipo incluyen programas de la categoría de los denominados "parsers" y son el

primer obstáculo en la simplicidad necesaria para la solución de ecuaciones por medio de microprocesadores.

Podemos simplificar muchísimo nuestra programación en el micro introduciendo, un nivel de representación inferior al de las definiciones alfanuméricas, codificando cada término de **SP *en binario***, colocando, por ejemplo, un "**0**" si la variable está **afirmada** y un "**1**" en caso de que aparezca **negada** (note que ésta, que es nuestra convención, resulta contraria a la de "Espresso" y "Spice" vistas arriba; pero el procedimiento se establece de tal manera que los resultados no cambian).

Así, nuestra **ecuación** podría definirse de la siguiente forma:

```
Out0 = 111 + 101 (Representación SP: 2 términos: SP1 y SP2)
```

en donde "`Out0`" representa el **bit0** del vector de resultados; es decir, que en esa posición es donde esperamos encontrar el valor resultante para la función `F(A, B, C)`. Si tuviéramos más ecuaciones, las siguientes serían "`Out1`", "`Out2`" y así sucesivamente, y ocuparía cada una los bits **bit1, bit2**, etc. del vector de resultados.

Obsérvese que también podemos expresar la ecuación en hexadecimal, para mayor comodidad pero, obviamente, las dos significan lo mismo:

```
Out0 = 0x7 + 0x5
```

Cada bit en "0" o en "1" de los que conforman los términos SP representa el valor que debe tomarse de las variables de entrada, ordenadas de izquierda a derecha como A, B, C, en nuestro ejemplo. Así se ve claramente que el término SP2: "`~A*B*~C`" estaría representado por "`101`" (el orden de las variables también es convencional; podría emplearse otro cualquiera sin que los resultados cambiaran, a condición de mantener la coherencia a través de todo el procedimiento).

Si pudiéramos almacenar las representaciones numéricas de los distintos términos de Suma de Productos SP, en la memoria del microcomputador, codificados en binario tal como acabamos de sugerir, para **resolver la ecuación** tendríamos que escoger, por cada bit de los diversos SP, *la correspondiente entrada,* ***afirmada*** *si el bit del SP está en "**0**", y* ***negada*** *si el bit vale "**1**".* Se haría el AND con los bits así seleccionados, y esto daría el valor de la función para cada término SP.

Recuérdese que la versátil función XOR puede verse como si fuera un "negador condicional"; esto es, un dispositivo que suministra, como salida, la entrada de información, **afirmada** si la otra señal, la condicional, tiene el valor "**0**", y produce la variable **negada** si la condicional vale "**1**". Éste es exactamente el comportamiento descrito *en letra itálica* en el párrafo anterior, ***y esta es la razón de***

por qué hemos hecho así la selección de nuestra convención, aunque resulte al revés de las demás...

El microprocesador haría un ciclo, posiblemente infinito, para leer cada vector de entrada **I** y, dentro de ese bucle, una iteración para procesar **cada uno** de los diversos componentes SP1, SP2, etc. del vector de términos **SP.** Si encuentra que para algún término de SP la evaluación vale "1", el microcomputador puede producir inmediatamente un "1" como resultado de esa ecuación. Sólo si se hubieran evaluado todos los términos SP sin haber encontrado jamás un "1", se produciría un "0" como valor calculado de la fórmula.

Esta solución puede verse esquematizada de la siguiente manera:

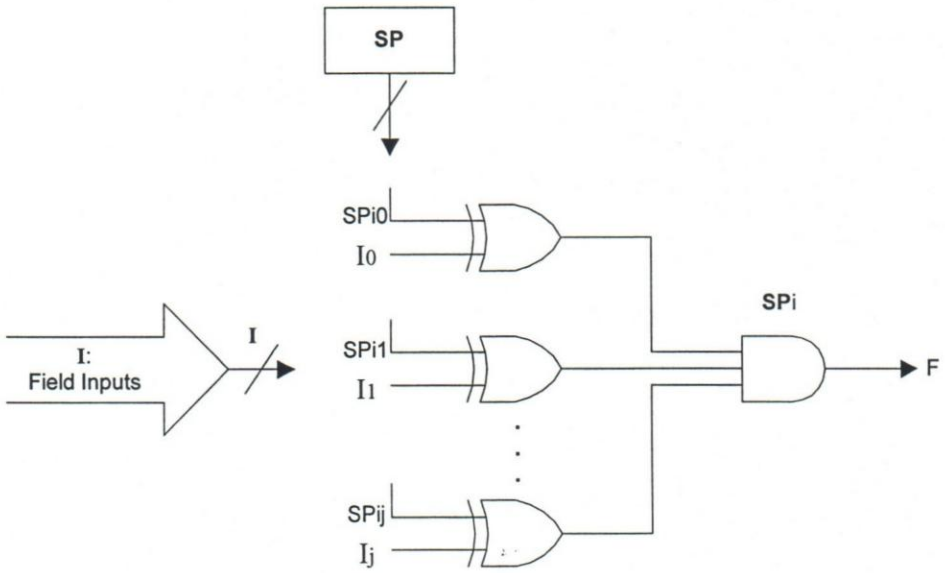


Fig. 2-2
Esquema explicativo del proceso de cada elemento canónico, almacenados en SP

En la parte superior de la Fig. 2-2 se representa la memoria en donde se almacenan todos los términos SP de la ecuación Canónica, codificados en binario tal como se indicó.

Si, por ejemplo, analizamos sobre la Fig. 2-2 el comportamiento del segundo término de **SP**: SP2 (`101`, con las variables ordenadas como A, B, C), el producto (AND) obtenido sería `~A*B*~C`, según corresponde a la ecuación: El primer término de "`101`"; como está en "1", hace pasar por el correspondiente XOR la variable `A` negada: `~A`; para el segundo término, que está en "0", pasa por su XOR la variable `B` afirmada, y para el último, que vale "1", la variable pasa negada: `~C`.

Salta a la vista que si la función la manejáramos simplificada, obtendríamos un menor número de términos de SP, y como el ciclo interno se hace para procesar **cada uno** de los componentes SP, el microcomputador tendría en promedio menos iteraciones por hacer.

➢ Ecuaciones No Canónicas, SP

Supongamos que tenemos la siguiente función normal, mínima, no Canónica, de A, B, C, expresada siempre como Suma de Productos:

```
F(A, B, C) = A*B + C
```

La diferencia fundamental con la función Canónica es que, por definición, en los términos SP de aquella aparecen siempre todas las variables; así que no es necesario identificar cuáles variables participan en ellos: todas lo hacen.

Cuando la ecuación es Normal, si quisiéramos aplicar un método similar al sugerido para las Canónicas nos encontraríamos en la necesidad de definir para cada término SP, cuáles variables aparecen en él. En otras palabras, cada término SP debe codificarse de tal manera que no solo nos permita saber si las variables figuran afirmadas o negadas sino que, además, identifique si una variable participa o no dentro del término.

Es decir, que a cada variable, que en la ecuación Canónica se representaba como un bit, hay que añadirle información adicional que señale si participa o no en cada término. Las posibles combinaciones para cada variable son entonces:

- **No**: No participa en el término SP
- **Sí**: Sí participa en el término SP **y**:
  - Se toma **Negada** o:
  - Se toma **Afirmada**

Nótese que no importa el valor de la variable si ésta no participa en algún término SP, como es lógico pensar.

Claramente se ve la necesidad de emplear al menos dos bits para definir ahora cada variable en los diferentes términos SP, puesto que hay tres condiciones que codificar para cada una de ellas. La asignación que se ha escogido para esos dos bits facilita muchísimo tanto la codificación de las ecuaciones, cuando hay que prepararlas para introducirlas al microcomputador, como la manipulación que internamente hace el mismo de las expresiones lógicas, como veremos a continuación.

Con todos estos elementos podemos avanzar y presentar al fin nuestra propuesta completa, comenzando primero con ***XBIN***, o codificación interna empleada para las ecuaciones booleanas en ***UBM***, y en seguida con ***UBM***, nuestra metodología específica para ecuaciones Normales, no canónicas.

Capítulo 3

XBIN, CODIFICACIÓN BINARIA EXTENDIDA

La representación XBIN de las ecuaciones es el corazón de UBM

Perspectiva

PARA obtener el ***XBIN***, o ***Codificación Binaria Extendida*** de una ecuación lógica, se forma una **Tabla Auxiliar para la Codificación,** con el agregado de las variables **afirmadas**, seguido a continuación por las variables **negadas** en idéntico orden. A cada columna se le asigna su equivalente binario clásico de acuerdo a su posición: **2^n**, n = 0, 1, 2,... Para el caso de tres variables A, B, C, la Tabla Auxiliar para la Codificación Binaria Extendida ***XBIN***, sería, por ejemplo:

```
       LH        |       RH        <= Left Half, Right Half, de SP
  32    16     8 |    4    2    1 <= Ponderación posicional binaria: 2^n
0x20  0x10   0x8 | 0x4  0x2  0x1 <= Ponderación posicional en hexadecimal
---------------|------------
  ~A    ~B    ~C |    A    B    C <= Agregado de variables negadas y afirmadas
```

Cada término SP se codifica **sumando** los aportes individuales determinados por la posición de cada variable, afirmada o negada. Por ejemplo, la siguiente función no Canónica, de ABC, en Suma de Productos:

```
F(A,B,C) = A*B + C
```

Tendrá como codificación ***XBIN*** la siguiente, según la anterior Tabla Auxiliar:

```
F(A,B,C) = 0x6 + 0x1
```

(**A** vale **4**, **B** tiene la ponderación **2**; el primer término resulta en **4 + 2**. El segundo término, **C**, es **1**. El signo "+", representando el OR, se mantiene)

Otro ejemplo. Si quisiéramos obtener el código ***XBIN*** de la función "XOR de dos variables A y B", el resultado sería el siguiente:

```
   XOR(A,B) = A*~B + ~A*B

  LH   | RH   <= Left Half, Right Half, de SP (almacenamiento Suma de Productos)
 8  4  | 2 1  <= Ponderación posicional binaria: 2^n
------|----
~A ~B  | A B  <= Agregado de variables (negadas y afirmadas) , para dos variables

   XOR(A,B) = 0x6 + 0x9
```

Recuérdese que en la codificación sugerida anteriormente para la solución de funciones en Forma Canónica en Suma de Productos, por cada bit de los diferentes términos SP tendría que escogerse la correspondiente entrada, **afirmada** si el bit del SP estaba en "**0**", o **negada** si el bit valía "**1**". Se haría el AND entre todos los bits así seleccionados, y esto daría el valor de cada SP.

En el caso de la codificación ***XBIN*** el comportamiento es ***distinto":*** para los diferentes términos SP, si el bit $\mathbf{SP_j}$ está en "0" la variable asociada, afirmada o negada, **no** participa en la expresión, y si $\mathbf{SP_j}$ vale "1" **sí** participa, afirmada o negada, dependiendo de con cuál valor de la variable (el afirmado o el negado, **LH** o **RH**) se encuentra asociado dicho $\mathbf{SP_j}$ en la Tabla Auxiliar para la Codificación (sea, . Se hace el AND con los bits así seleccionados y esto produce el valor de la función para cada término de SP. ***¡Así de fácil!***

Es importante resaltar que, como los bits seleccionados son las entradas a una función AND, ***si una variable no forma parte de un término, su bit asociado SP_j deberá introducirse como un "1" al AND*** *(Entradas no usadas en una compuerta AND se colocan en "1" y en un OR se conectan a "0").* Y si participa, pues se lleva la variable de entrada I (o ~I), con el valor que tenga. Más precisamente, cada bit que participa en el AND se define mediante las siguientes expresiones:

$b_j = \sim SP_j + I_j$, para la mitad de SP asociada con las variables afirmadas, ó
$b_k = \sim SP_k + \sim I_k$, para la mitad de SP asociada con las variables negadas

De esta formulación puede observarse claramente que los términos SP ***sólo aparecen negados***; podría por tanto sugerirse que se generaran directamente los términos negados en la codificación ***XBIN***. Sin embargo, lo que en realidad se

hace es generarlos afirmados como ya se indicó y se los ***almacena negados*** en la memoria del microcomputador.

Podemos esquematizar la solución anterior de la siguiente manera:

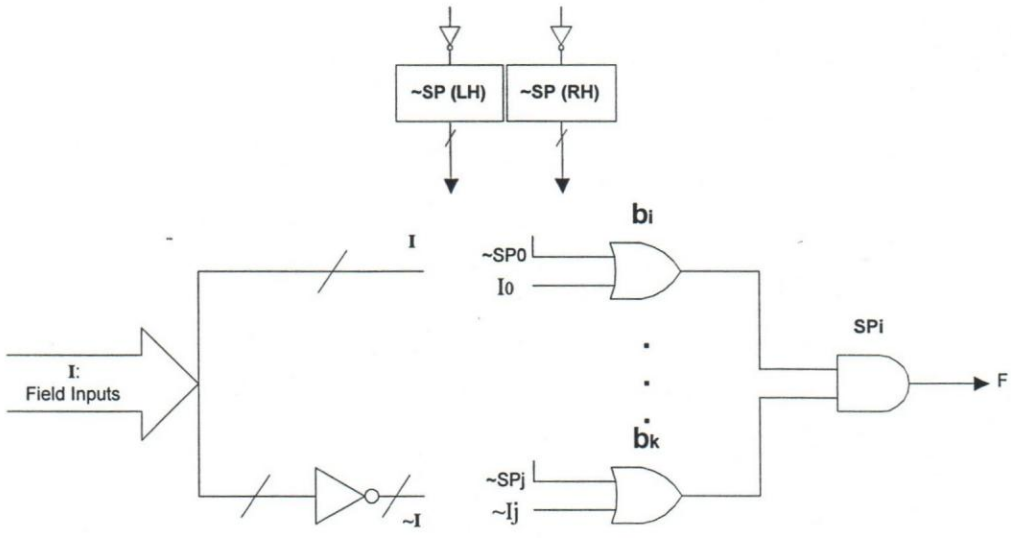


Fig. 3.1

En la parte superior de la fig. 3.1 se observa representada aquella parte de la memoria del microprocesador en donde se guardarán las dos mitades de cada uno de los términos de Suma de Productos SP en codificación ***XBIN***; las hemos llamado mitad derecha RH e izquierda LH (**R**ight and **L**eft **H**alf); se ven también unos negadores a la entrada de la memoria, para indicar que los códigos ***XBIN*** se almacenan ***negados***, puesto que negados es que habrá de empleárselos, según acabamos de señalar.

Para el término de producto SPi, se muestra en la figura cómo cada bit del resultado está representado por: $b_j = \sim SP_j + I_j$ ó $b_k = \sim SP_k + \sim I_k$, y todos los bits se procesan, finalmente, mediante el operador AND.

Con una codificación tan elegante como la de ***XBIN***, a ***UBM*** le queda un trabajo muy sencillo: El microcomputador establece un ciclo, probablemente infinito, para leer cada vector de entrada **I**; dentro de él ejecuta otra iteración en la que procesa cada uno de los componentes SP1, SP2, etc. del vector de términos **SP.** Cuando encuentre que para algún SP la evaluación del respectivo AND vale "1", se produce inmediatamente un "1" como respuesta de la ecuación. Sólo si se evaluaran todos los términos **SP** sin encontrar jamás un "1", el resultado que produciría la fórmula sería un "0".

Capítulo 4

UBM, MÁQUINA BOOLEANA UNIVERSAL

El algoritmo.

LA fig. 3.1 muestra el poderoso y sencillo corazón de *UBM* para un bit del resultado (*UBM1*); es decir, para resolver **una** ecuación lógica, formada por una cantidad no restringida de variables de entrada. En la práctica se programa, por ejemplo, para 16 ecuaciones (16 bits de resultado) y para 16 variables de entrada. Para otros valores como 32 u 8 hay que adaptar el programa, pero esto consiste en una actividad que resulta muy sencilla y directa.

Si una aplicación tiene *menos* entradas que las programadas, debe garantizarse que sus respectivos bits lleguen al microcomputador como "ceros". Además, las salidas con los valores resultantes pueden ser impredecibles para los bits que no han sido definidos como salidas mediante ecuaciones.

Otro valor para el que hay que establecer un tope máximo es para el número de términos SP en cada ecuación; suelen usarse no más de 256 productos por función pero en la práctica esta cantidad puede ser mucho menor. Este es un parámetro ajustable dentro del programa y sólo depende de la cantidad de memoria que se encuentre disponible en el microprocesador.

Por comodidad se asume que todos los bits de entrada están alambrados en forma consecutiva, a partir del **bit0** en adelante. Lo mismo ocurre con las salidas: arrancan a partir del bit0. Las entradas no empleadas deben ir a "0" y a las salidas no definidas **no** debe conectarse nada en el micro. Es muy importante esta última restricción para las salidas, ya que si hay realimentaciones (cfr. Capítulo 5), el programa las asume como salidas virtuales y las coloca a la izquierda de las salidas verdaderas; por eso es imperativo que no se conecten jamás al mundo exterior (excepto en el caso en que las realimentaciones formaran en realidad una parte deseada del vector de salida).

Si asumimos que se han almacenado previamente dentro del microcomputador las ecuaciones apropiadamente codificadas, el ***algoritmo UBM1****, (UBM para 1 bit) reflejado en la fig. 3.1 para un bit de resultado (una sola ecuación) hasta de 16 variables de entrada y un número de términos SP acotado sólo por la capacidad de memoria del micro, puede resumirse de la siguiente manera:*

- Se lee cada vez el vector de entrada `I` (16 bits)
- Se genera un vector de entrada extendido, formado por `~I | I` (32 bits); es decir, el agregado (`|`) de los bits de `I` y de `~I`
- Se toman uno a uno los términos `~SP` para la ecuación y se hace cada vez el OR de todos sus bits con el vector de entrada extendido: `~I | I`
  - Se reduce el resultado a un valor lógico de verdad (`Vn`) mediante la siguiente regla:
    `SI( ~SPn | I )` tiene TODOS los bits en "1", => finalice el proceso e inmediatamente:
    `retorne Vn = 1`
    De lo contrario, `Vn = 0`; continúe el proceso con los demás términos `~SP`
  - Si al finalizar todos los `SP` ningún `Vn` dio "1" como resultado, `retorne Vn = 0`

Esto es todo: ① una lectura de la entrada: `I`, ② la negación y la extensión de dicha entrada: `~I | I`, ③ un ciclo con un "bit OR" entre los valores consecutivos de `~SP` y la entrada extendida: `~I | I`, ④ una comparación y ⑤ el retorno del resultado, conforman el algoritmo suficiente para evitarnos un "parser", el Diagrama de Escaleras ("Ladder Logic") o el programa, cada vez diferente y específico, en Basic, "C", o mediante algún método "ad-hoc" que pudiera estar disponible.

El algoritmo UBM1, para vectores de entrada hasta de 16 bits, escrito en pseudo "C", es el siguiente:

```
I |= ~I << 16;        // ① y ②: Generar vector de entrada extendido
for( i = 0; i < n ; i ++ ) {       // ③: Tomar uno a uno los ~SP
    V = SP[ i ] | I;               // ..y se hace "bit OR" con I
    if( r = (V == 0xFFFFFFFFL) )   // ④: "r" almacena resultado
        break;                     // Todo "1"s? Retorne "1"
}
return( r );   // ⑤: Retorne el bit resultante para la ecuación
```

La variable "n" indica el número de términos `SP` que tiene la ecuación.

El Algoritmo UBM16, para un máximo hasta de 16 ecuaciones, tiene la siguiente estructura (**equ_index** es el número actual de ecuaciones):

```
for( i = 0; i < equ_index ; i ++ ) {
    R |= Ubm1( input, SP[ i ], NSP[ i ] )  <<  i;
}
return( R );     // R: 16 bits Boolean Result for ALL equations
```

`SP [ i ]` indica dónde comienzan los términos `SP` para la presente ecuación, y `NSP[ i ]` es la cantidad de dichos términos.

Muy sencillo: un ciclo que recorre todas las ecuaciones (`i < equi_ndex`), en el cual se llama a la rutina central ***UBM1***, una vez por cada ecuación, pasándole como parámetros la entrada actual, la dirección donde comienzan los términos SP de dicha ecuación y la cantidad de tales términos. El resultado se va ensamblando (`R |= Ubm1() << i`) bit a bit en el vector de resultados `R`, hasta de 16 elementos. De esta manera, como se indicó con anterioridad, el `bit0` de `R` corresponde al resultado de la primera ecuación, `R1` al segundo, y así sucesivamente.

Algoritmo para UBM-Combinatorio

Al programa principal le bastaría con establecer el ciclo infinito de adquisición de las variables de entrada, llamar a ***UBM16*** y entregar el resultado, de la siguiente forma:

```
Forever                 // Ciclo sin fin
   I = inp();           // Se lee la entrada I (16 bits)
   R = Ubm16( I );      // Calcula hasta 16 ecuaciones
   out(R);              // Entrega hasta 16 bits resultantes
Endforever
```

Capítulo 5

UBM EN SISTEMAS SECUENCIALES

El tiempo.

SI se establecen **realimentaciones**, el algoritmo se hace genérico y permite procesar, no solo los casos combinatorios, sino también los secuenciales [4]. Esto se hace de una manera muy sencilla: se generan como salidas "internas", o "virtuales", las realimentaciones, y se las vuelve a introducir en las ecuaciones como las demás entradas. No es ninguna novedad, ya que así es que se diseñan redes secuenciales asíncronas.

Algoritmo Genérico para UBM, Combinatorio y Secuencial

```
R       = initR;       // Valor inicial de Resultados
FeedMsk = initFM;      // "0" si no hay realimentación
Shift   = initSh;      // A partir de dónde es el feedback

Forever                 // Ciclo sin fin
   I = inp();           // Se lee la entrada I (16 bits)
   I |= (R & FeedMsk ) << Shift;
                        // _^_: Realimenta de R (valor ACTUAL
                        // ..de Resultados) a I
   R = Ubm16( I );      // Calcula valor FUTURO de R
                        // ..con un máximo de 16 ecuaciones
   out(R);              // Entrega valor calculado de R
                        // ..hasta 16 bits resultantes
EndForever
```

[4] Para mantener la simplicidad de la explicación, en el código hasta aquí expuesto se han obviado las indispensables definiciones de las variables.

En este algoritmo, la expresión añadida por sobre el procedimiento para redes combinatorias:

```
I |= (R & FeedMsk ) << Shift; // FEEDBACK variables, de R a I
```

toma el vector `R` de Resultado anterior, le elimina los bits que son salidas **externas** (la salidas reales), mediante el "`&`" con la máscara `FeedMsk`; desplaza `R` hasta posicionarlo en donde comienzan las salidas "internas" (realimentaciones), vía "`<< Shift`" y finalmente lo superpone con las verdaderas entradas provenientes del campo, a través de un bit OR: "`I |=`".

El algoritmo ***UBM*** secuencial es general porque de él puede derivarse el combinacional, puesto que para este caso basta con hacer que la máscara `FeedMsk` valga "0" (y, por *congruencia*, `Shift` también deberá valer "0").

Este método de establecer las realimentaciones asume que las variables **internas** (feedback) se sitúan a continuación de las variables reales.

UBM se limita estrictamente a la resolución de ecuaciones booleanas, tanto combinatorias como secuenciales. Si el dispositivo de control ("embedded processor") precisa hacer uso de otros mecanismos internos, muy usados, tales como temporizadores, "Schmitt triggers", comparadores con valores analógicos, etc., como se esbozó en la presentación del ejemplo inicial, deberán ser programadas rutinas apropiadas por separado y estos valores de entrada han de incorporarse dentro de la rutina principal de ***UBM***, que debe recibir la información correspondiente de estos objetos internos e insertar sus valores digitales en los bits apropiados del vector de entrada `I` (alrededor de la instrucción: `I = inp()` del algoritmo genérico. La rutina `inp()` puede ser vista como un método genérico de adquirir, no solo un conjunto de bits externos, sino también como la encargada de acopiar el resto de la información (interna) que se precise y presentarla apropiadamente dentro del vector de entrada).

Capítulo 6

HERRAMIENTAS DE SOPORTE

Soporte en el Internet

ANEXO al presente documento va un programa de demostración, para línea de comandos de Windows: **GenCSeq**, del cual se suministran las fuentes, y que resuelve tanto redes combinatorias como secuenciales cuando se invoca con las siguientes opciones:

```
USAGE:
GenCSeq InitR  FeedMsk  Shift  Equations.file  Inputs.file >Out


ALL input 16 bit numerical values may be: normal Decimal 1-9NN,
Hex (0xNN), Octal (0NN) or binary (b:NN) numbers.

For COMBINATORIAL logic, FeedMsk = 0 (Shift = 0)
```

Como puede verse todos los parámentros se indican en la línea de comando: el valor inicial de R, `InitR`, la máscara para la realimentación, `FeedMsk` y la cantidad de posiciones donde ésta debe comenzar, `Shift`; la designación de las ecuaciones mediante un archivo de entrada `Equations.file`, lo mismo que los valores que asumen las variables, para efecto de la demostración: `Inputs.file`.

Se anexa también el programa **MakeXBin.pl**, escrito en lenguaje "Perl", por si el usuario se decide a no codificar manualmente sus expresiones. **MakeXBin.pl** toma un conjunto de ecuaciones lógicas, y produce automáticamente los códigos en *XBIN*, en un formato tal que puede emplearse directamente en **GenCSeq**. **MakeXBin.pl** incluye su propia documentación.

Los programas pueden obtenerse accediendo con el Explorador a:

`https://webstorage.btinet.net/`

Cuando la página abra, puede entrar con los siguientes parámetros:

E-mail: `LGUribe@telcel.net.ve`

Password: **ubm**
(minúsculas)

Ir a la carpeta **UBM** (es la única carpeta) y entrar con doble click.

Allí encontrará un archivo: "**UBM.zip**" (sólo hay un archivo) que contiene:

- **GenCSeq.c**: Ejemplo de un: "Boolean Engine Evaluator, for 16 bits" (Programa en C)
- **GenCSeq.exe**: Ejecutable del anterior programa, para Windows (línea de comandos)
- **MakeXBin.pl**: Make Xbin: eXtended Binary Code (Programa en PERL)

En caso de cualquier inconveniente para obtener los programas indicados, favor enviar un mensaje a:

`GUribeC@cantv.net`

GenCSeq

```
#include "ezc.h"

ID    =
"GenCSeq.c - Generic COMBINATORIAL / SEQUENTIAL\n"
"     ABM, A Boolean Machine Evaluator, for 16 bits\n"
"  Luis G. Uribe C., V1.21 D05G1 M23S2 S28D2 J09E3 S11E3\n\n";

USAGE =
" USAGE: GenCSeq InitR FeedMsk Shift Equations.file  Inputs.file >Out\n"
"        ALL input (16 bits) numerical values may be: normal decimal\n"
"        ..(1-9NN), Hex (0xNN), Octal (0NN) or binary (b:NN) numbers\n"
"        For COMBINATORIAL logic, FeedMsk = 0 (Shift = 0)\n";

#if 0
" A) Formulate the problem in Sum of Products (SP) equations;
"
" B) Get their 'Xbin' representation, formed by filling "1" for each
"    variable that is used from the EXTENDED SP Vector (aggregate of
"    asserted variables catenated with their negated values, for each
"    SP). Each of this binary equivalent corresponds with each Sum of
"    Products term, SP0, SP1,..., SPn.
"    You may use 'MakeXBin.pl' Perl program to preprocess the
"    equations. It will produce the 'Xbin' representation for you.
"
" NOTE: 'MakeXBin.pl -s' takes non ordered equations and produces
"    the appropiate Extended Numerical Boolean Equivalents.
"
" NOTE: 'MakeXBin.pl -i -s -f' is the most common used command.
"    Please see 'MakeXBin.pl' documentation.
#endif


/* ------------------------------------------------------------- */
/* Include Files                                                 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* ------------------------------------------------------------- */
/* Defines                                                       */

#define N_EQ       16      // Max number of equations
#define MAX_SP    256      // Max number of Sum Of Products Terms
#define LSIZE    4096      // Max size (chars) of input lines
```

```
/* ------------------------------------------------------------ */
/* Global Variables                                             */
/* NOTE: sizeof(SP) = N_EQ * MAX_SP * 4 (16*256*4= 16Kbytes)    */
/*       Take this into account, in order to preserve DATA space...*/

FILE  *eq, *finp;          // Input files: Equations and Field inputs
char   linebuf[ LSIZE ];   // Input lines buffer; must include \n & \0
ulong  SP[ N_EQ ][ MAX_SP ];  // 32 bit Sum of Products for equations
                           // ..(16 bits) aggregated to their 16
                           // ..negated values  /// V1.2, J09E3

int    equindex = 0;       // index to SP; Number of Equations
int    NSP[ N_EQ ];        // Number of SP terms in each equation


/* ------------------------------------------------------------ */
/* Function Prototypes                                          */

bool   Ubm1(  uint I, ulong SP[], int n );     /// V1.21, S11E3
uint   Ubm16( uint input );
char  *ltrim ( char *buf ); // trim leading spaces; remove '\n'
void   usage(  int exerr );
char  *xtobin ( uint d );
ulong  Xstrtoul( char *argv );


/* ============================================================ */
void   main ( int argc, char **argv )                    /*()*/
Begin      /* GenCSeq.c */

   /* ------------------------------------------------------------ */
   /* LOCAL VARIABLES                                              */

   // R : BOOLEAN 16 BITS RESULT VECTOR FOR EQUATIONS. Init as needed
   // ..for sequential circuits. 0 for Combinatorial

   uint  R;


   // FeedMsk : Mask to select output bits to be Fedback, as next
   // ..inputs, via: I |= ( R & FeedMsk ) << Shift; If you need to
   // ..fedback more than one bit, they MUST be consecutive.
   // EXAMPLE : Feedback 2 bits begining on bit #3 of Output, begining
   // ..on bit #5 of next input:
   //       FeedMsk = (1<<4 | 1<<3 ); Shift = 5;
   // ..If you need to feedback bits #2 and #4, starting on bit #5:
   //       FeedMsk = (1<<4 | 1<<2 ); Shift = 5;
   // ..Will paste b40b2 begining on next read input, at bits b70b5

   uint  FeedMsk;


   // Shift : Places to shift left the selected feedback bits before
   // ..oring them with Input
```

```
uint  Shift;


// I : Input 16 bits boolean vector. You *MUST* assure that non
// ..used bits on input will be 0 for the "I |=" feedback from
// ..output into input to work!

uint  I;


//------------------------
int   i, j;             // Loop variables
bool  input = FALSE;    // Detect empty input file
bool  ineq  = FALSE;    // '=' sign detected (INside EQuation)
char *p;                // Auxiliar char pointer


/* ============================================================ */
/* Verify input parameters                                      */

If( argc < 6 )
    fprintf( stderr,
        "\nError -1: argc '%d' must be >= 6\n", argc );
    usage( -1 );
Endif

/* ------------------------------------------------------------ */
/* Get Initial R                                                */

++ argv;
-- argc;
R = (uint)Xstrtoul( *argv );

/* ------------------------------------------------------------ */
/* Get FeedMsk                                                  */

++ argv;
-- argc;
FeedMsk = (uint)Xstrtoul( *argv );

/* ------------------------------------------------------------ */
/* Get Shift ammount                                            */

++ argv;
-- argc;
Shift = (uint)Xstrtoul( *argv );


/* ------------------------------------------------------------ */
/* Read all Equations (SP) from the first file: 'equ.fil'       */
/* INPUT FILE FORMAT: is produced by 'MakeXBin.pl -s -f',       */
/*  as follows:                                                 */
```

```
#if 0
   #Dummy = P*O*N*M*L*K*J*I*H*G*F*E*D*C*B*A     # Set variables order
   #Dummy =
   #0x0000FFFF;

   #R1 = A*~B + ~A*B;
   R1 =
   0x00020001 +
   0x00010002;

   #R2 = A*B;
   R2 =
   0x00000003;

   #R3 = A + B;
   R3 =
   0x00000001 +
   0x00000002;
#endif

   ++ argv;
   -- argc;
   If( ( eq = fopen( * argv, "rb" ) )  is  NULL )
       fprintf( stderr,
           "\nError -2: unable to open '%s' file\n", * argv );
       usage( -2 );
   Endif

   i = 0;
   While( fgets( linebuf, LSIZE, eq )  dif  NULL )
       ltrim( linebuf );   // trim line leading spaces; remove '\n'
       If( *linebuf  is  '#' )
           continue;       // Discard lines begining with # (comments)
       Endif

       If( not *linebuf )                  // skip empty lines
           continue;                       // next line...
       Endif

       If( strstr( linebuf, "=" ) )        // Begin of Equation
           ineq = TRUE;
           continue;                       // next line...
       Endif

       input = TRUE;
       SP[ equindex ][ i ++  ] =
           bnot Xstrtoul( linebuf );       /// Store negated, V1.2, J09E3

       If( ineq )
           If( strstr( linebuf, ";" ) ) // End of Equation?
               ineq = FALSE;
               NSP[ equindex ] = i;    // Number of SP terms in this eq
               i = 0;
               equindex ++;            // point to next equation
```

```
        Endif
    Else
        fputs( "\nError -3: 'ineq' is FALSE", stderr );
        exit( -3 );
    Endif
Endwhile
fclose( eq );

If( not equindex )          // Need at least one Equations defined
    fprintf( stderr,
        "\nError -4: no valid equations found on '%s' file\n",
        * argv );
    usage( -4 );
Endif

/* ------------------------------------------------------------ */
/* Print Command line params; identify Equations & Inputs files */

p = * argv;

++ argv;
-- argc;
If( ( finp = fopen( * argv, "rb" ) )  is  NULL )
    fprintf( stderr, "\nError -5: file '%s' not found\n", * argv );
    usage( -5 );
Endif

printf( "       GenCSeq Parameters:\n"
    "InitR = '0x%04X', FeedMsk = '0x%04X', Shift = '0x%04X'\n\n",
    R, FeedMsk, Shift );

/* ------------------------------------------------------------ */
/* Print previous read input SP & inverted SP terms             */

printf( "Extended Sum of Products Terms "
    "(Equations file = '%s')\n", p );
For( i = 0; i < equindex; i ++ )
    printf( "Eq[%2d] = ", i );
    For( j = 0; j < NSP[ i ]; j ++ )
        printf( "0x%08lX, ", ~SP[ i ][ j ] );   /// V1.2, J09E3
    Endfor
    putchar( '\n' );
Endfor
putchar( '\n' );


/* ============================================================ */
/* **ALGORITHM**: Apply boolean equations --defined through SP  */
/* ..elements--, to each Input vector taken from the Input.fil  */

printf( "Consecutive input vectors (file '%s'), "
        "and binary Results:\n", * argv );
printf( "        R = %s <= Initial Value\n", xtobin( R ) );
input = FALSE;
```

```
While( fgets( linebuf, LSIZE, finp )  dif  NULL )
    ltrim( linebuf );           // trim leading spaces; remove \n
    If( *linebuf  is  '#' )     // skip comments (begining with #)
        continue;
    Endif
    If( not *linebuf )
        continue;                   // skip empty lines
    Endif

    input = TRUE;                   // mark input file as NOT empty

/* ------------------------------------------------------------ */
/* C) Read each input vector I, formed by the aggregate of bits */
/*    of the (up to) 16 normally affirmated variables           */
/*    (AB in the example above)                                 */
/*                                                              */
/* NOTE: You *MUST* assure that input non used bits will be 0   */
/*    ..for the "I |=" feedback from output into input to work! */

    I = (uint)( Xstrtoul( linebuf ) );

    ////////////////////////////////////////////////////////////
    //  At this point we could  FEEDBACK  some previous outputs //
    //  from R into I, in order to make a Sequential Circuit.   //
    ////////////////////////////////////////////////////////////

    I |= ( R band FeedMsk )  <<  Shift;     // Feedback variables

    printf( "0x%04X, ", I );        // Read 'I' pasted w/feedback

    R = Ubm16( I );


    /* ------------------------------------------------------- */
    /* Print Results (in binary for demostration purposes)     */

    printf( "R = %s\n", xtobin( R ) );

Endwhile    // While( fgets( linebuf, ...

If( not input )
    fprintf( stderr,
        "\nError -6: no valid input data found on '%s' file\n",
        * argv );
    exit( -6 );
Endif

exit( 0 );                      // automatically close 'finp' file

End         /* main() */
```

```
/* ************************************************************** */
uint   Ubm16( uint input )                                        /*()*/
Begin      /* Ubm16() */

   /* ************************************************************ */
   /* INTERFACE WITH * Ubm16() * FUNCTION:                         */
   /*                                                              */
   /*   For each (up to) 16 input bits I, produce one 16 bits      */
   /*   output boolean results, R                                  */
   /*                                                              */
   /* INPUTS                                                       */
   /*   uint input : 16 bits (MUST be) Input vector for this       */
   /*                program; other values (8, 32) must be         */
   /*                programed following this schema               */
   /*                                                              */
   /* GLOBAL INPUT VARIABLES USED                                  */
   /*   ulong SP[] : Vector with  ~ Sum of Products EXTENDED Terms */
   /* int equindex : Number ID (index) of the equation             */
   /*    int NSP[] : Number of SP terms in each equation           */
   /*                                                              */
   /* OUTPUT                                                       */
   /*       uint R : one full 16 bits vector of boolean results    */
   /*                                                              */
   /*         First  equation result goes on bit 1 << 0,           */
   /*         Second equation result goes on bit 1 << 1,           */
   /* Generically, N equation result goes on bit 1 << n-1 (0..15)  */
   /* ************************************************************ */

   /* ------------------------------------------------------------ */
   /* LOCAL VARIABLES                                              */

   uint R = 0;             // one 16 bits vector of boolean results
   int  i;                 // Loop variable

   /* ============================================================ */
   /* Very compact Ubm16 algorithm                                 */

   For( i = 0; i < equindex ; i ++ )
       R |= Ubm1( input, SP[ i ], NSP[ i ] )  <<  i;
   Endfor
   return( R );            // Boolean Result for the equation

End        /* Ubm16() */
```

```
/* ============================================================== */
bool   Ubm1( uint input, ulong SP[], int n )                  /*()*/
Begin      /* Ubm1() */

   /* ************************************************************ */
   /* INTERFACE WITH * Ubm1() * FUNCTION:                          */
   /*                                                              */
   /*  For each (up to) 256 SP terms of one logical equation       */
   /*  produce One bit boolean result value, r                     */
   /*                                                              */
   /* INPUTS                                                       */
   /*  uint input : MUST be 16 bits Input vector on this program   */
   /*  ulong SP[] : Vector with  ~ Sum of Products Extended Terms  */
   /*       int n : Number of Sum of Products in SP                */
   /*                                                              */
   /* NO GLOBAL INPUT VARIABLES ARE USED                           */
   /*                                                              */
   /* OUTPUT                                                       */
   /*      bool r : One bit boolean result value                   */

   /* ------------------------------------------------------------ */
   /* LOCAL VARIABLES                                              */

   ulong I;                // Input 32 bits vector: ~input|input
                           // (| means 'aggregated')
   int   i;                // Loop variable
   ulong V;                // Boolean Value of Sum Of Products N: SPn
   bool  r = FALSE;        // Boolean Result for the equation


   /* ============================================================ */
   /* D1) First take the Input Vector (16 bits)                    */

   I = (ulong)input;

   /* ------------------------------------------------------------ */
   /* D2) Form the EXTENDED Input Vector through the catenation of */
   /*     ~I|I   (...~B~A | ...B A) (32 bits)                      */

   I = (ulong)input;
   I |= ~I << 16;          // Equations & inputs *MUST* be 16 bits
                           // ..vectors for *this* program to work

   /* ------------------------------------------------------------ */
   /* E) With the EXTENDED Input vector I make an evaluation,      */
   /*    taking ALL '~SPn' terms of equation, ORing them with I,   */
   /*    and reducing each OR result to a boolean value, using the */
   /*    following rule:                                           */
   /*                                                              */
   /*      If( ~SPn  bitOR  I ) has ALL bits "1", => Value Vn = 1  */
   /*      Else: Value Vn = 0                                      */
   /*                                                              */
   /*    Return 1 immediatelly with first Vn that takes on the "1" */
```

```
   /*    value.                                                    */
   /*    Return 0 if *NO* Vn took on the "1" value for some SP.    */

   I = (ulong)input;
   I |= ~I << 16;          // Equations & inputs *MUST* be 16 bits
   For( i = 0; i < n ; i ++ )      // 32 bits arithmetic for 16 boolean
       V = SP[ i ]  bor  I;        // ..input variables /// V1.2, J09E3
       If( r = ( V == 0xFFFFFFFFL ) )
           break;                  // first V = 1 is enough for Result
       Endif
   Endfor
   return( r );    // Return Boolean 1 bit Result for this equation

End        /* Ubm1() */


/* ================================================================ */
/* Ancillary subroutines for GenCSeq.c                              */

/* ---------------------------------------------------------------- */
char  *ltrim ( char *buf )  // trim leading spaces; remove \n  /*()*/
Begin      // Trim Leading spaces ON SITE

   char *p;

   if( buf[ strlen( buf ) ? strlen( buf ) - 1 : 0 ] is '\n' )
       buf[ strlen( buf ) ? strlen( buf ) - 1 : 0 ]  = '\0';
   if( buf[ strlen( buf ) ? strlen( buf ) - 1 : 0 ] is  '\r' )
       buf[ strlen( buf ) ? strlen( buf ) - 1 : 0 ]  = '\0';

   p = buf + strspn( buf, " \t\n\r" );
   if( p  dif  buf )
       strcpy( buf, p );

   return( buf );

End        /* trim() */

/* ---------------------------------------------------------------- */
void   usage( int exerr )                                    /*()*/
Begin      /* usage() */

    fputs( _id, stderr );
    fputs( _usage, stderr );
    exit( exerr );

End        /* usage() */
```

```
/* ------------------------------------------------------------- */
#define    MARK ( (1<<12) bor (1<<8) bor (1<<4) )      // 3 spaces

char  *xtobin ( uint d )                                    /*()*/

   /* 07-Oct-84, Luis G. Uribe C., toma un 'uint' y genera   *
    * un string con 16 0's y 1's binarios que corresponden   */

Begin   /* xtobin() */

   static char buf[ 20 ];          // 20: 16 bits + 3 spaces + '\0'
   char *bp;
   uint  msk;

   For( ( bp = buf, msk = (uint)1<<15 ); msk; msk >>=1 )
          * bp ++ = (char)( ( d band msk )  dif  0 ) + '0';
       if( msk  band  MARK )   *bp++ = ' ';
   Endfor
   * bp = '\0';
   return( buf );

End     /* xtobin() */


/* ------------------------------------------------------------- */
ulong  Xstrtoul( char *argv )                               /*()*/
Begin      /* Xstrtoul() */    // eXtend 'strtoul()' with "b:"

   char *p;

   If( ( p = strstr( argv, "b:" ) )  or
       ( p = strstr( argv, "B:" ) ) )
       return( strtoul( p + 2, NULL, 2 ) );
   Else
       return( strtoul( argv,  NULL, 0 ) );
   Endif

End        /* Xstrtoul() */

#if 0
" STRTOUL: If base is 0, the initial characters of the string
" pointed to by nptr are used to determine the base.
" - If the first character is 0 and the second character is not
"   'x' or 'X', then the string is interpreted as an OCTAL integer;
" - otherwise, it is interpreted as a decimal number.
" - If the first character is '0' AND the second character is 'x'
"   or 'X', then the string is interpreted as a hexadecimal integer.
" - If the first character is '1' through '9', then the string is
"   interpreted as a decimal integer. The letters 'a' through 'z'
"   (or 'A' through 'Z') are assigned the values 10 through 35; only
"   letters whose assigned values are less than base are permitted.
#endif
```

MakeXBin.pl

```
#!/usr/bin/perl

=head1 NAME: MakeXBin.pl

 Make Xbin: eXtended Binary Code: Take Boolean equations and form the
 numerical equivalent expressions suitable to be processed by Abe16
 (A Boolean Engine) or GenCSeq.exe

              Luis G. Uribe C., M24D2 C15E3

=head1 SYNOPSIS


 perl MakeXBin.pl [-i] [-d] [-s] in >out
  -i: ignore case (default is no ignore case)
  -f: skip First equation (default is no skip)
  -d: print debug information (default is no debug)
  -s: Single SP in each line  (default is all SP terms in a line)

=cut

#---------------------------------------------------------------------
# Default 'use' setings

use warnings;
use strict;

#---------------------------------------------------------------------
# Constants and Parameters

use constant LVALUE  => 1 << 0;
use constant RVALUE  => 1 << 1;
use constant LRVALUE => LVALUE | RVALUE;
use constant FALSE   => "";
use constant TRUE    => (! FALSE);

#---------------------------------------------------------------------
# Configuration Variables and default values; modify from command line

our $icase  = 0;    # -c: ignore case
our $debug  = 0;    # -d: print debug information
our $single = 0;    # -s: Single SP in each line
our $fskip  = 0;    # -f: skip First equation


#---------------------------------------------------------------------
# External Libraries

use Getopt::Std;
```

```
#-------------------------------------------------------------------
# Program Variables

my( @equations, %vars, $line, $scolon, $bit );
my  %opts;          # command line parameters from getopts


#===================================================================
# Load command line parameters, optionally

getopts( "dfis", \%opts );

#-------------------------------------------------------------------
# -c: ignore case (default is no ignore case)

$icase  = 1 if defined $opts{ "i" };

#-------------------------------------------------------------------
# -d: print Debug information (default is no print)

$debug  = 1 if defined $opts{ "d" };

#-------------------------------------------------------------------
# -f: skip First equation (default is no skip)

$fskip  = 1 if defined $opts{ "f" };

#-------------------------------------------------------------------
# -s: print Debug information (default is no print)

$single = 1 if defined $opts{ "s" };


#*******************************************************************
# MAIN
# read input file and split into one line EQUATIONS

while(<>) {
   chomp;
   s/#.*$//;                # remove comments
   s/\s+/ /g;               # trim spaces
   s/\s*$//;                # remove trailing spaces
   next if /^\s*$/;         # skip empty lines
   undef $scolon;
   $line .= $_ if $_;       # append
   next unless /;/;         # continue appending following lines

   $scolon = 1;             # found semicolon ';': store each equation
   push @equations, $line;
   die "wrong number of '=|;'; probably mixing equations on same line"
       if $line =~ /=.*=|;.*;/;
   undef $line;
}
```

```
die "missing <;>" if ! defined $scolon;

#-----------------------------------------------------------------
# store VARIABLES

foreach ( my @leq = @equations ) {
   s/\s+//g;                # remove spaces
   s/~//g;                  # remove negates
   s/'//g;                  # remove negates
   my @tvars = split/[^0-9A-Za-zÁÉÍÓÚáéíóúÜüÑñ=]/; # Spanish \W
   foreach ( @tvars ) {
       next if /^\s*$/;         # skip empty lines
       $_ = lc( $_ ) if $icase;
       if( /(.*)=(.*)$/ ) {     # Res '=' Var ?
           $vars{$1}{LR} |= LVALUE;
           $vars{$2}{LR} |= RVALUE;
           $vars{$2}{BIT} = $bit ++ if !defined $vars{$2}{BIT};
       }else{
           $vars{$_}{LR} |= RVALUE;
           $vars{$_}{BIT} = $bit ++ if !defined $vars{$_}{BIT};
       }
   }
}
warn "bit number ($bit) > 16" if $bit > 16;     # for 16 bits...

# /@{[LVALUE]}/: How interpolate a CONSTANT inside a string or regex:
# Remember: CONSTANTs are implemented as anonymous subroutines!
#
# The way it works is that when the @{...} is seen in the double-quoted
# string, it's evaluated as a block. The block creates a reference to
# an anonymous array containing the results of the call to
# mysub(1,2,3). So the whole block returns a reference to an array,
# which is then dereferenced by @{...} and stuck into the double-quoted
# string.

if( $debug ) {
   foreach my $var ( sort { lc($a) cmp lc($b) } keys %vars ) {
       print "# $var, ";
       SWITCH: foreach ( $vars{$var}{LR} ) {
           /@{[LVALUE]}/  and print( "LVALUE" ), last;
           /@{[RVALUE]}/  and print( "RVALUE" ), last;
           /@{[LRVALUE]}/ and print( "BOTH: LVALUE and RVALUE" ), last;
           print "# bad variable $var => <$_>";
       }
       if( defined $vars{$var}{BIT} ) {
           print "\n# $var BITID = $vars{$var}{BIT}\n";
       }else{
           print "\n";
       }
   }
}

#-----------------------------------------------------------------
# store negated VARIABLES (~var)
```

```
foreach my $var ( sort { lc($a) cmp lc($b) } keys %vars ) {
   if( $vars{$var}{LR} & RVALUE ) {
       $vars{"\~$var"}{BIT} = $vars{$var}{BIT} + $bit;
       $vars{"\~$var"}{LR} |= RVALUE;
       $vars{"$var'"}{BIT} = $vars{$var}{BIT} + $bit;
       $vars{"$var'"}{LR} |= RVALUE;
       if( $debug ) {
           print "# $var' BITID = ".$vars{"\~$var"}{BIT}.", LR = ";
           SWITCH: foreach ( $vars{"\~$var"}{LR} ) {
               /@{[LVALUE]}/  and print( "LVALUE" ), last;
               /@{[RVALUE]}/  and print( "RVALUE" ), last;
               /@{[LRVALUE]}/ and print( "BOTH: LVALUE and RVALUE" ), last;
               print "# bad variable $var => <$_>";
           }
           print "\n";
       }
   }
}

#-------------------------------------------------------------------
# split EQUATIONS into Sum Of Products terms and PRINT:

foreach my $eq ( @equations ) {
   my $result;
   $eq =~ s/\s+//g;            # remove spaces
   $eq =~ s/;//g;              # remove ;
   my @sp = split/\+/, $eq;
   foreach ( @sp ) {
       my( $tmp, @var );
       $_ = lc( $_ ) if $icase;
       if( /(.*)=(.*)/ ) {
           $result .= $1 . " = ";
           $result .= "\n" if $single;
           @var = split/\*/, $2;
       }else{
           @var = split/\*/;
       }

       foreach ( @var ) {
           $tmp |= 1 << $vars{$_}{BIT};
       }

       if( ! $fskip ) {
           $result .= sprintf( "0x%08X + %s", $tmp, $single ? "\n" : "" );
       }else{
           $result .= sprintf( "#0x%08X + %s", $tmp, $single ? "\n" : "" );
       }
   }
   $result =~ s/ \+ $//;
   $result =~ s/\n$//s;

   $eq =~ s/([=\+])/ $1 /g;
```

```
    print "#$eq;\n";
    if( ! $fskip ) {
        print "$result;\n\n";
    }else{
        print "#$result;\n\n";
        $fskip = FALSE;
    }
}

#**********************************************************************
# POD documentation

=head1 DESCRIPTION

Ubm16 - Universal Boolean Machine, or its GenCSeq.c, -Generic
Combinatorial/Sequential Boolean Engine Evaluator-, solve boolean
equations in generic ways, so we don't need to make a program each
time we have a digital problem to be solved using a generic CPU in
embedded applications, for example. Those programs need, however,
that the input boolean equations be fed coded as 'Xbin': eXtended
Binary Code. This is a special numerical encoding technique that
facilitates the programing of the UBM Evaluators.

MakeXBin.pl is a Perl program to code logical equations into Xbin:
eXtended Binary Code, i.e. it takes Boolean equations and form the
eXtended Binary equivalent expression.

The input file contains the boolean equation(s) to be encoded,
written in free format. You may use spaces to make more readable your
expressions, and break equations in several lines. For this to
succeed, equations need to be terminated with a ';' sign. You may
not, however, mix several equations in the same line.

 For example:

 R = ~B*A + B*~A;        # A XOR B, function of PONMLKJIHGFEDCBA.

 #alternatively:

 R = ~ B * A +
     B*~A;               # A XOR B, function of PONMLKJIHGFEDCBA.

 #Wrong, the following won't work:

 R1 = ~ B * A; R2 = X*Y*Z;

 Accepted symbols to form equations are:
 ---------------------------------------

=over 4

=item  '=' sign: Assignment
```

The equal sign, as usual, breaks the boolean equation into a left and a right side. The result of the expression defined to the right is stored in the variable located at the left side.

=item '~' sign: Negation

The negation symbol is placed at the left of the variable that it applies.

=item '*' sign: AND

The AND sign is placed between two variables.

=item '+' sign: OR

The OR sign is placed between two Sum of Products (SP) terms.

=item ';' sign: Terminator

The semicolon is the required equation terminator. It terminates the boolean expression. Remember anyway that even if you can split equations through several lines, a new one need to begin in its own line.

=item '#' sign: Comments sign

Every thing written before the '#' sign is not taken into account. You may write comments in 2 flavors:

```
# This is a comment line

R = A*B;    # This is a comment
```

=item 'Variables': Alfanumeric identifiers

The normal boolean variables used in digital equations. Normally it make differences between upper and lower cases, but you may override this through the -i switch.

=back

Note that the expressions must be written in Sum of Products terms. They don't need to be minimized, or sorted by any concept, but you cannot use parenthesis or Product of Sums terms. Just plain SP.

=head2 XBIN: EXTENDED BINARY CODE

It consists in assigning a position to each variable, beginning from the right (bit 0), and proceeding to the left, in increasing bit order (bit 1, bit 2, etc). It also assigns a position to the negated variables, following the position of the last asserted variable. This produces an eXtended vector with the asserted and the negated

```
variables: ~I|I.

The numerical representation is standard binary coding, only that the
asserted variables take as a value: 2**m (m being their place on
the vector) and the negated variables take also values: 2**n (n being
the place of the negated variable in the extended vector)

 In this way see how the following SP terms are encoded:

 A    is coded as: 0x1
 ~A   is coded as: 0x2 and
 A*~A is coded as: 0x3.

 If we had had 2 variables, A and B, the following will be coded:

 A   is coded as: 0x1,
 B   is coded as: 0x2,
 ~A  is coded as: 0x4 and
 ~B  is coded as: 0x8.
 So, A*~B is is coded as: 0x9.

 .

=head2 CODING TRICKS

MakeXBin.pl assigns sequential ordering to the variables in the way
of their first apparition. At end of file, it assign now position for
the negated variables, at the next free place left from the
asserted ones, following identical order that those variables,
i.e., ~C~B~ABCA (from left to right)

If you would need to make a fixed length (say, 16 bits), i.e., you
need the first 16 bits to be asserted (that is typical), you can
help yourself defining a dummy equation in the first place, as
follows:

 dummy = P*O*N*M*L*K*J*I*H*G*F*E*D*C*B*A;   # 16 variables
 R = ~B*A + B*~A;         # A XOR B, function of PONMLKJIHGFEDCBA.

 The program will produce the following code:

 dummy = 0x0000_FFFF;
 R = 0x0002_0001 + 0x0001_0002;

 instead of:

 R = 0x0000_0009 + 0x0000_0006;
 that would be given if only the variables A and B exist:

 #Input:
 R  = ~B*A + B*~A;         # A XOR B, function only of BA
 AA =  A;
 BA =  B;
 AN = ~A;
```

```
 BN = ~B;

 #Output:
 R  = 0x0000_0009 + 0x0000_0006;
 AA = 0x0000_0001;
 BA = 0x0000_0002;
 AN = 0x0000_0004;
 BN = 0x0000_0008;

=head1 OPTIONS

=over 4

=item -i: ignore case (default is no ignore case)

Usefull when writting sequential equations where the same variable is
written in upper and lower case, i.e: Q = q*~K + ~q*J;

=item -f: skip First equation (default is no skip)

Should you like to state some fixed order for the variables, and their
ammount, you may write a dummy equation, as follows:

 Dummy = K*J*Q          # 16 Input Variables
        *D04*D05*D06*D07*D08*D09*D10*D11*D12*D13*D14*D15*D16;

 Q = q*~K + ~q*J;       # Ignore Case (MakeXBin.pl -i -s -f )

Using the -f option, the first, dummy equation is commented out, as
follows:

 #Dummy = K*J*Q*D04*D05*D06*D07*D08*D09*D10*D11*D12*D13*D14*D15*D16;
 #dummy =
 #0x0000FFFF;

 #Q = q*~K + ~q*J;
 q =
 0x00010004 +
 0x00040002;

 .


=item -d: print debug information (default is no debug)

Print out the order of each variable, the number of variables, the
order for the negated variables.

=item -s: Single SP in each line  (default is all SP terms in a line)

-s option will provide a file that is more easy to parse. Usefull
for feeding data to the programs that make the UBM Evaluation (i.e.:
GenCSeq.exe, UBM16.EXE, etc.)
```

```
=back

=head1 REQUIRE

Perl 5.6

=head1 AUTHOR

Luis G. Uribe C., <LGUribe@cantv.net>

=head1 COPYRIGHT and LICENSE

Copyright (c) 2002 Luis G. Uribe C., (LGUribe@cantv.net). All
rights reserved.
```

Capítulo 7

EJEMPLOS

Un ejemplo vale más que mil demostraciones

CON el programa **GenCSeq** pueden repetirse en un PC los siguientes ejemplos; recomendamos experimentar con él y adaptar de ahí las rutinas que forman el núcleo de *UBM* para su uso específico.

Ejemplo 1: T (Implicit Clocked) Flip Flop:

La ecuación que define un flip flop tipo **T** (toggle, implicit clock) es:

```
Q = q*~T + ~q*T;        # Ignore Case (MakeXBin.pl -i -s -f )
```

La codificación *XBIN*, supondiendo que hay 16 variables de entrada (que obviamente no aparecen en esta sola ecuación) y que iría en el archivo `T-ff.eq`, es:

```
q = 0x00010002 + 0x00020001;
```

Si ejecutamos el programa:

```
GenCSeq.exe 0 1 1 T-ff.eq T-ff.in >T-ff.res
```

en donde puede apreciarse que el valor inicial del resultado R es "0", que la máscara de realimentación corresponde a un "1", ya que el primer bit del resultado, y el único en este caso, es el mismo que se va a realimentar, y que el número de posiciones que hay que desplazar este bit para que se integre con las variables realmente leídas es "1"; para quedar de esta forma a la derecha de la única entrada física, T.

Si el archivo `T-ff.in` contiene la siguiente serie de valores (dados en binario):

```
# T (implicit clock) Flip-Flop
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
```

se producen los siguientes resultados:

```
       GenCSeq Parameters:
InitR = '0x0000', FeedMsk = '0x0001', Shift = '0x0001'

Extended Sum of Products Terms (Equations file = 'T-ff.eq')
Eq[ 0] = 0x00010002, 0x00020001,

Consecutive input vectors (file 'T-ff.in'), and binary Results:
        R = 0000 0000 0000 0000 <= Initial Value
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0000
```

Obsérvese que la salida Q del flip-flop T es el bit0 del vector R de resultados (primera columna de la derecha), que se encuentra en binario (16 bits). Esta señal alternativamente vale "1" y "0" (toggle), ya que la entrada T está permanentemente en "1". Nótese también que la primera columna de la izquierda, que representa las entradas en hexadecimal, no solo aparece el sempiterno "1" de la entrada externa, sino que también se sobrepone la salida "q" realimentada.

Note que las ecuaciones codificadas en XBIN se terminan con un ";" para ser correctamente procesadas por el programa GenCSeq.exe. Esto permite introducir los términos, uno por cada línea de entrada y, en general, separarlos por espacios en formato libre, a fin de facilitar su lectura. La ecuación puede suministrársele a GenCSeq.exe como:

```
q = 0x00010002 + 0x00020001; y también como:
q =
0x00010002 +
0x00020001;
```

El ";" separa unas ecuaciones de las otras.

Ejemplo 2: SRCp Flip Flop (Explicit Clocked):

La ecuación que define un flip flop tipo SRCp (Set, Reset, Clock pulse) es:

```
Q = S*Cp + q*~R + q*~Cp;
# Ignore Case: MakeXBin.pl -i -s -f
```

La codificación ***XBIN***, supondiendo que hay 16 variables de entrada y que iría en el archivo `SRCp-ff.eq`, es:

```
q = 0x00000003 + 0x00040008 +
0x00010008;
```

Si ejecutamos el programa:

```
GenCSeq.exe 0 1 3 SRCp-ff.eq
   SRCp-ff.in >SRCp-ff.res
```

Aquí vemos de nuevo que el valor inicial del resultado R se asume como "0", que el bit "1" del resultado (y único) ha de realimentarse, para lo cual hay que desplazarlo tres posiciones, con lo cual quedará a la izquierda de S, R y Cp.

Si el archivo de entrada `SRCp-ff.in` contiene la siguiente serie de valores (dados en binario):

```
# SRCp Flip-Flop
b:000
b:001
b:000

b:010
b:011
b:010

b:100
b:101
b:100

b:000
b:001
b:000
```

en donde las entradas, de izquierda a derecha, son S, R y Cp, en ese orden. Nada debe ocurrir mientras Cp vale "0". Set hará que la salida del flip flop vaya a "1" y Reset la llevará a "0".

Se producen los siguientes resultados:

```
       GenCSeq Parameters:
InitR = '0x0000', FeedMsk =
'0x0001', Shift = '0x0003'

Extended Sum of Products Terms
(Equations file = 'SRCp-ff.eq')
Eq[ 0] = 0x00000003, 0x00040008,
0x00010008,

Consecutive input vectors (file
'SRCp-ff.in'), and binary Results:
        R = 0000 0000 0000 0000 <=
            Initial Value
0x0000, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
0x0002, R = 0000 0000 0000 0000
0x0003, R = 0000 0000 0000 0001
0x000A, R = 0000 0000 0000 0001
0x000C, R = 0000 0000 0000 0001
0x000D, R = 0000 0000 0000 0000
0x0004, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
```

Note que hasta la quinta entrada no sucede ningún cambio en la salida; verifique que es lo correcto. Igualmente transcurren tres entradas más, antes de que la salida regrese a "0".

Ejemplo 3: JK (Implicit Clocked) Flip Flop:

La ecuación que define un flip flop tipo JK (implicit clock) Flip-Flop es:

```
Q = q*~K + ~q*J;
# Ignore Case: MakeXBin.pl -i -s -f
```

La codificación ***XBIN***, suponiendo que hay 16 variables de entrada y que iría en el archivo **JK-ff.eq**, es:

```
q = 0x00000003 + 0x00040008 +
0x00010008;
```

Si ejecutamos el programa:

```
GenCSeq.exe 0 1 2 JK-ff.eq
   JK-ff.in >JK-ff.res
```

Vemos que el valor inicial del resultado R se asume como "0", que el bit "1" del resultado (y único también) ha de realimentarse, para lo cual hay que desplazarlo dos posiciones, con lo cual quedará a la izquierda de J y K.

Si el archivo de entrada **JK-ff.in** contiene la siguiente serie de valores (dados en binario):

```
# JK (implicit clock) Flip-Flop
b:00
b:01
b:00

b:10
b:00

b:01
b:00

b:11
b:11
b:11
b:11
```

(Esta tabla de entradas se refieren a las variables J y K, en ese orden, de izquierda a derecha).

Se producen los siguientes resultados:

```
       GenCSeq Parameters:
InitR = '0x0000', FeedMsk =
'0x0001', Shift = '0x0002'

Extended Sum of Products Terms
(Equations file = 'JK-ff.eq')
Eq[ 0] = 0x00010004, 0x00040002,

Consecutive input vectors (file
'JK-ff.in'), and binary Results:
        R = 0000 0000 0000 0000 <=
            Initial Value
0x0000, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
0x0002, R = 0000 0000 0000 0001
0x0004, R = 0000 0000 0000 0001
0x0005, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
0x0003, R = 0000 0000 0000 0001
0x0007, R = 0000 0000 0000 0000
0x0003, R = 0000 0000 0000 0001
0x0007, R = 0000 0000 0000 0000
```

Observe que hasta el 4 grupo de entradas no cambia a "1" la salida del flip flop; permanece dos tiempos ahí y luego se va por dos más a "0". Finalmente entra en un estado de "toggle".

Ejemplo 4: Flip Flop Set-Reset sin Reloj:

La ecuación que define un flip flop tipo SR sin reloj es la clásica:

```
Y = S + ~R*y;  # Ignore Case (MakeXBin.pl -i -s -f )
```

La codificación *XBIN*, supondiendo que hay 16 variables de entrada y que iría en el archivo `ff.eq`, es:

```
y = 0x00000001 + 0x00020004;
```

Si ejecutamos el programa:

```
GenCSeq.exe 0 1 2 ff.eq ff.in >ff.res
```

y el archivo de entrada `ff.in` contiene la siguiente serie de valores (dados en binario):

```
# RS Flip-Flop, sin reloj
b:0000
b:0001
b:0000
b:0010
b:0000
```

Se producen los siguientes resultados:

```
       GenCSeq Parameters:
InitR = '0x0000', FeedMsk = '0x0001', Shift = '0x0002'

Extended Sum of Products Terms (Equations file = 'ff.eq')
Eq[ 0] = 0x00000001, 0x00020004,

Consecutive input vectors (file 'ff.in'), and binary Results:
        R = 0000 0000 0000 0000 <= Initial Value
0x0000, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0004, R = 0000 0000 0000 0001
0x0006, R = 0000 0000 0000 0000
0x0000, R = 0000 0000 0000 0000
```

Ejemplo 5: Múltiples Entradas Combinatorias:

Verificar el comportamiento de las siguientes ecuaciones combinatorias:

```
#R00 = ~B*~A;
R00 = 0x00030000;

#R01 = ~B*A;
R01 = 0x00020001;

#R02 = B*~A;
R02 = 0x00010002;

#R03 = B*A;
R03 = 0x00000003;

#R04 = ~B*~A + ~B*A;
R04 = 0x00030000 + 0x00020001;

#R05 = ~B*~A + B*~A;
R05 = 0x00030000 + 0x00010002;

#R06 = ~B*~A + B*A;
R06 = 0x00030000 + 0x00000003;

#R07 = ~B*A + B*~A;
R07 = 0x00020001 + 0x00010002;

#R08 = ~B*A + B*A;
R08 = 0x00020001 + 0x00000003;

#R09 = ~B*A + ~B*~A;
R09 = 0x00020001 + 0x00030000;

#R10 = B*~A + B*A;
R10 = 0x00010002 + 0x00000003;

#R11 = B*~A + ~B*~A;
R11 = 0x00010002 + 0x00030000;

#R12 = B*~A + ~B*A;
R12 = 0x00010002 + 0x00020001;
```

Ejemplo 6: 3 bit counter (Implicit Clocked):

Las ecuaciones que definen un contador binario de 3 bits son las siguientes:

```
# Ignore Case: MakeXBin.pl -i -s -f
Q1 = q1* ~T  + ~q1 *T;
Q2 = q2* ~(q1*T) + ~q2* (q1*T);
Q3 = q3* ~(q2*q1*T)
   + ~q3* (q2*q1*T);
```

Que pueden escribirse en Suma de Productos (eliminando los paréntesis) como:

```
# Ignore Case: MakeXBin.pl -i -s -f
Q1 = q1* ~T  + ~q1 *T;
Q2 = q2* ~q1 + q2*~T + ~q2*q1*T;
Q3 = q3* ~q2 + q3*~q1 + q3*~T
   + ~q3*q2*q1*T;
```

La codificación *XBIN*, supondiendo que hay 16 variables de entrada y que iría en el archivo `3bcnt.eq`, es:

```
#Q1 = q1*~T + ~q1*T;
q1 = 0x00010002 + 0x00020001;

#Q2 = q2*~q1 + q2*~T + ~q2*q1*T;
q2 = 0x00020004 + 0x00010004 +
     0x00040003;

#Q3 = q3*~q2 + q3*~q1 + q3*~T
#    + ~q3*q2*q1*T;
q3 = 0x00040008 + 0x00020008 +
     0x00010008 + 0x00080007;
```

Si ejecutamos el programa:

```
GenCSeq.exe 0 7 1 3bcnt.eq
   3bcnt.in >3bcnt.res
```

y el archivo de entrada `3bcnt.in` contiene la siguiente serie de valores (dados en binario):

```
# 3 Flip-Flops counter; input:
Count
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
b:1
```

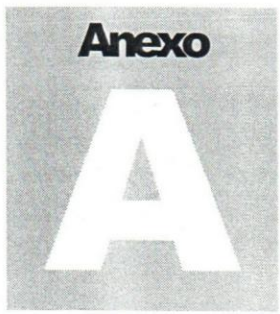
Se producen los siguientes resultados:

```
       GenCSeq Parameters:
InitR = '0x0000', FeedMsk =
'0x0007', Shift = '0x0001'

Extended Sum of Products Terms
(Equations file = '3bcnt.eq')
Eq[ 0] = 0x00010002, 0x00020001,
Eq[ 1] = 0x00020004, 0x00010004,
         0x00040003,
Eq[ 2] = 0x00040008, 0x00020008,
         0x00010008, 0x00080007,

Consecutive input vectors (file
'3bcnt.in'), and binary Results:
        R = 0000 0000 0000 0000 <=
            Initial Value
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0010
0x0005, R = 0000 0000 0000 0011
0x0007, R = 0000 0000 0000 0100
0x0009, R = 0000 0000 0000 0101
0x000B, R = 0000 0000 0000 0110
0x000D, R = 0000 0000 0000 0111
0x000F, R = 0000 0000 0000 0000
0x0001, R = 0000 0000 0000 0001
0x0003, R = 0000 0000 0000 0010
```

Lo que representa un contador binario de tres flip flops, que cuenta de 0 a 7 (en el ejemplo regresa otra vez a 0 y continúa hasta 2).



"RELAY LADDER LOGIC PROGRAMMING" EN EL PLC

Habiendo ya dicho suficiente sobre la Programación de Escalera, vamos a redondear ahora la idea mostrando el funcionamiento desde el punto de vista del otro extremo: El PLC. Miraremos, como ejemplo, una sección del código residente en un PLC comercial basado en un PC industrial, con un microprocesador Intel de la familia 80x86 y el sistema operativo DOS. Nuestra intención no es profundizar en él, sino presentarlo sólo como ilustración, para incitar la curiosidad de los lectores.

El programa es del tipo TSR (Terminate and Stay Resident) para el viejo sistema operativo DOS, pero es representativo de lo que ocurre en los PLCs aún hoy en día. Este tipo de programa comienza ejecutando las instrucciones finales (la etiqueta *Start* en el ejemplo), y cuando termina y queda residente desaloja de la memoria el código de inicialización, que ya no se necesitará más. Ese comienzo en el programa que presentaremos tiene una parte de inicialización de variables y parámetros (con algunas secciones indicadas sólo a nivel de comentario); determina a continuación si no ha sido residenciado con anterioridad, y procede a instalar el mecanismo de comunicación tanto con el "timer" (**TimerHook:**) como con el segmento que ejecuta el código compilado por la estación de trabajo a partir del diagrama de escalera (**LadderHook:**). Hecho lo anterior, deja residentes en memoria las partes recurrentes del código.

Las estructuras de datos más significativas son: Un bloque de memoria (2000 bytes en el ejemplo) reservado a partir de la posición identificada como "**WorkPad**"; allí es donde deberá alojarse con posterioridad el programa producido para el PLC por la estación de trabajo, el cual puede ser **a**) código orientado directamente al micro (compilado) o, **b**) lo que es más común, un programa generado para una "máquina virtual", con una arquitectura que cada manufacturante de PLCs define y que deberá ser *emulada* por el micro; este último es el caso de nuestro actual ejemplo. Los comandos del Interpretador incluyen operaciones básicas entre dos operandos: AND, OR, XOR y NOT, además de instrucciones para LOAD y STORE de valores (o el equivalente MOVE). En nuestro programa de ejemplo, la máquina objeto del código que debe interpretarse asigna un operando (A) a uno de sus acumuladores internos, el **AL** del 80x86, y el otro operando (B) está en alguna posición de memoria correspondiente a las variables.

Las demás estructuras de datos del programa están conformadas por las variables de entrada (**VarArrayINP**), que se identifican con tres símbolos alfanuméricos: Una **I** mayúscula (**input**) y dos números, que en el ejemplo llegan sólo hasta "**I07**". A continuación de esos 3 bytes se almacena el valor de la variable de entrada propiamente dicho (un byte, codificado en binario, probablemente en complemento a 2 o en exceso 128). De manera similar se define el conjunto de variables de salida (VarArrayOUT, hasta "**O03**" en el ejemplo, también con valores de un byte). Hay un grupo de temporizadores (**VarArrayTIMER**, para un total de 4 en nuestro caso), que van llevando el tiempo de

diferentes eventos; asociado con ellos está un grupo de valores que determinan el estado que habrán de asumir los contadores cuando deban ser recargados (**VarArrayRELOAD**). Sigue un conjunto de variables denominadas "**coils**" (**VarArrayCOIL**, 16 en el ejemplo), y que representan entradas (contactos) y salidas binarias de control (cada coil representa un bit, y corresponde a un relé). Mencionamos por último un grupo de variables que sirven para llevar la hora de ciertos eventos; son los relojes.

En realidad pueden existir muchos más componentes en un PLC moderno, como los relés internos, que no son ni entradas ni salidas físicas, sino variables locales; relés con retardo (monostables), ya vistos en los ejemplos de programación de escalera; contadores, unidades de procesamiento para realizar operaciones aritméticas o comparar cantidades numéricas (EQ, NEQ, LES, GRT, etc). También suelen tener capacidad para control local (PID) y funciones: Aritméticas, trigonométricas, raíces, logaritmos... [5]

Obsérvese que en la sección "**LadderHook**" aparece una secuencia importante:

```
call    GetInput
call    GetClock            ; update CLOCK status
call    WorkPad             ; run logic equations
call    UpdateOutput
```

muy similar a la que usamos con anterioridad para ilustrar la Aproximación Convencional (ver arriba en la sección de código: RCP_digital)

Note por último que, dependiendo del tamaño del equipo PLC, las funciones de Programación de Escalera, que corresponden a la Estación de Trabajo, y la parte de control propiamente dicha, que es la del PLC, ***pueden encontrarse incorporadas en el mismo equipo***.

A continuación, extractos del programa de ejemplo, residente en el PLC.

[5] Hay una estandarización para el lenguaje "Relay Ladder Logic Programming", que puede consultarse en las referencias al final de este documento.

```
page 60,132

.model small
.code
.386

T_Hook          equ 80h             ; User defined vectors
L_Hook          equ 81h
C_Hook          equ 82h
T_HookZP        equ T_Hook * 4
L_HookZP        equ L_Hook * 4
C_HookZP        equ C_Hook * 4
OpcodeRET       equ 0C3h            ; opcode for 'RET' instruction
InputPort       equ 379h            ; This equipment uses parallel printer
OutputPort      equ 378h            ; ..ports for field input/output
PSP             equ 10h

ProgramID       db  'TEST'          ; ID String to verify program resident

;---------------------------------------------------------------------
; Execute Relay Ladder Logic Programming via INT 81h (L_Hook)
; ON ENTRY: nothing; 'I' flag copied from stack while in this ISR.
; ON EXIT:  all inputs read, all coils and outputs updated.
; DESTROYS: nothing
;---------------------------------------------------------------------

LadderHook:
        push    bp
        mov     bp,sp
        push    [bp+6]
        popf                        ; assume 'IF' of calling program
        pusha
        push    ds
        mov     ax,cs
        mov     ds,ax               ; DS == CS

        call    GetInput
        call    GetClock            ; update CLOCK status
        call    WorkPad             ; run logic equations
        call    UpdateOutput

        pop     ds
        popa
        pop     bp
        iret

;---------------------------------------------------------------------
; Update TIMER00-TIMER03 and set corresponding byte in VarArrayTIMER
;    via INT 80h (T_Hook).
; ON ENTRY: nothing; 'I' flag copied from stack while in this ISR.
; ON EXIT:  TIMER(nn) skipped, reloaded or decremented,
;           all T(nn) updated
; DESTROYS: nothing
;---------------------------------------------------------------------
TimerHook:
        push    bp
        mov     bp,sp
        push    [bp+6]
        popf                            ; assume 'IF' of calling program
        pusha
        push    ds
        mov     ax,cs
        mov     ds,ax                   ; DS == CS
        mov     cx,4                    ; this code uses 4 Timers
        mov     si,offset VarArrayTIMER + 3
        mov     di,offset VarArrayRELOAD + 3
        mov     bx,offset TIMER00
```

```
Tim001:
       cmp     byte ptr ds:[di],0FFh   ; see if RELOAD coil set
       jne     Tim002
       mov     ax,word ptr ds:[bx+2]   ; get Reload value
       mov     word ptr ds:[bx],ax     ; and copy to TIMER
       mov     dl,0FFh
       cmp     ax,0
       je      Tim004                  ; Reload value=0, set TIMER coil & exit
       mov     dl,0
       jmp     Tim004                  ; Reload value>0, clear TIMER coil & exit
Tim002:
       cmp     word ptr ds:[bx],0
       mov     dl,0FFh
       je      Tim004                  ; TIMER done, set TIMER coil & exit
       mov     dl,0                    ; still decrementing, clear TIMER coil
       mov     ax,word ptr ds:[bx]     ; decrement, BCD
       dec     al
       das
       jns     Tim003                  ; only sec & min affected
       mov     al,59h                  ; also hours affected
       clc
       xchg    ah,al
       dec     al
       das
       xchg    ah,al
Tim003:
       mov     word ptr ds:[bx],ax     ; update, BCD
Tim004:
       mov     byte ptr ds:[si],dl     ; update TIMER coil
       add     bx,4                    ; next TIMER register
       add     si,4                    ; every four bytes in array
       add     di,4                    ; every four bytes in array
       loop    Tim001
       pop     ds
       popa
       pop     bp
       iret

;---------------------------------------------------------------------
; Fill VarArrayINP with data from Digital Inputs via 'call GetInput'
; ON ENTRY: DS points to VarArray segment
; ON EXIT:  all inputs read into VarArray.
; DESTROYS: AX, CX, DX, SI, Flags
;---------------------------------------------------------------------

GetInput:
       mov     cx,8                    ; 8 Digital Inputs
       mov     si,offset VarArrayINP + 3
       mov     ah,1                    ; start with Bit 0
       mov     dx,InputPort
       in      al,dx                   ; global capture
Get001:
       test    al,ah                   ; test bit
       mov     byte ptr ds:[si],0      ; if AL(8-CX) = 0
       jz      Get002
       mov     byte ptr ds:[si],0FFh   ; if AL(8-CX) = 1,2,4,..,128
Get002:
       shl     ah,1                    ; next bit position
       add     si,4                    ; every four bytes in array
       loop    Get001
       ret
```

```
;---------------------------------------------------------------------
; Update Digital Outputs from VarArrayOUT via 'call UpdateOutput'
; ON ENTRY: DS points to VarArray segment
; ON EXIT:  all outputs updated from VarArray
; DESTROYS: AX, BX, CX, DX, SI, Flags
;---------------------------------------------------------------------

UpdateOutput:
       mov     cx,4                    ; 4 Digital Outputs
       mov     si,offset VarArrayOUT + 3
       mov     ah,1                    ; start with Bit 0
       mov     al,0                    ; initial value
Upd001:
       mov     bl,byte ptr ds:[si]     ; get output value (0, 0FFh)
       and     bl,ah                   ; isolate bit position
       or      al,bl                   ; add to output byte
       add     si,4                    ; every four bytes
       loop    Upd001
       mov     dx,OutputPort
       out     dx,al                   ; global update
       ret

;---------------------------------------------------------------------
; Get Clock data from [C_Hook], apply to CLOCK(nn) and
;    update VarArrayCLOCK, via 'call GetClock'
; ON ENTRY: DS points to VarArray segment
; ON EXIT:  VarArray updated.
; DESTROYS: AX, BX, CX, SI, DI, Flags
;---------------------------------------------------------------------

GetClock:
       push    ds
       push    0
       pop     ds
       mov     bx,word ptr ds:[C_HookZP]   ;get BCD time (0000 to 2359)
       pop     ds
       cmp     bx,0
       je      Gck004                  ; not updated, do nothing....
       mov     cx,8
       mov     si,offset CLOCK00
       mov     di,offset VarArrayCLOCK
Gck001:
       cmp     word ptr ds:[si],bx
       jg      Gck002
       mov     al,byte ptr ds:[si+2]   ; ON/OFF
       mov     byte ptr ds:[di+3],al   ; update contact
       add     si,3
       loop    Gck001
Gck002:
       mov     cx,8
       mov     si,offset CLOCK01
       mov     di,offset VarArrayCLOCK + 4
Gck003:
       cmp     word ptr ds:[si],bx
       jg      Gck004
       mov     al,byte ptr ds:[si+2]   ; ON/OFF
       mov     byte ptr ds:[di+3],al   ; update contact
       add     si,3
       loop    Gck003
Gck004:
       ret
```

```
;-------------------------------------------------------------------
VarArrayStart  equ $
VarArrayINP    db  'I00',0, 'I01',0, 'I02', 0, 'I03', 0
               db  'I04',0, 'I05',0, 'I06', 0, 'I07', 0

VarArrayOUT    db  'O00',0, 'O01',0, 'O02', 0, 'O03', 0

VarArrayTIMER  db  'T00',0, 'T01',0, 'T02', 0, 'T03', 0

VarArrayRELOAD db  'R00',0, 'R01',0, 'R02', 0, 'R03', 0

VarArrayCOIL   db  'C00',0, 'C01',0, 'C02', 0, 'C03', 0
               db  'C04',0, 'C05',0, 'C06', 0, 'C07', 0
               db  'C08',0, 'C09',0, 'C10', 0, 'C11', 0
               db  'C12',0, 'C13',0, 'C14', 0, 'C15', 0

VarArrayCLOCK  db  'K00',0, 'K01',0
VarArrayEnd    equ $

TIMER00        dw  0
RELOAD00       dw  0
TIMER01        dw  0
RELOAD01       dw  0
TIMER02        dw  0
RELOAD02       dw  0
TIMER03        dw  0
RELOAD03       dw  0

CLOCK00        db  0, 0, 0             ; Slot #1, ON/OFF
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0             ; Slot #8, ON/OFF

CLOCK01        db  0, 0, 0             ; Slot #1, ON/OFF
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0
               db  0, 0, 0             ; Slot #8, ON/OFF

;-------------------------------------------------------------------
; Workpad is where all compiled PLC logic is stored for run-time
;    evaluation. Call via 'call Workpad'
;    Init Workpad storage area filled with 'RET Opcode'
;    Main program must fill WorkPad properly before calling it.
; ON ENTRY: DS points to VarArray segment
; ON EXIT:  all coil and output equations resolved and
;           written to VarArray
; DESTROYS: all registers, Flags
;-------------------------------------------------------------------

WorkPad:
        db      2000 dup(OpcodeRET)
```

```
;---------------------------------------------------------------------
; Program Start
;---------------------------------------------------------------------

Start:
; PUT CODE HERE to Scan string, something like:
       mov     al,'R'
       mov     bx,'20'
       call    ScanVarArray
; ...

; PUT CODE HERE to Get Parameters & Equations file from host PC
; ...

; THE FOLLOWING CODE IS STANDAR FOR DOS
; Replace vector for INT(T_Hook) with CS:TimerHook
       push    0
       pop     ds
       lds     si,ds:[L_HookZP]
       cmp     dword ptr ds:[si-4],'TSET'    ; 'TEST' in backwards
       jne     st001
       jmp     T_HookAlreadyInstalled
st001:
       push    cs
       pop     ds
       mov     dx,offset TimerHook
       mov     ah,25h
       mov     al,T_Hook
       int     21h

; Replace vector for INT(L_Hook) with CS:LadderHook
       mov     dx,offset LadderHook
       mov     ah,25h
       mov     al,L_Hook
       int     21h

; INSERT CODE HERE to set TIMER00-TIMER03 Reload values
; ...

; Terminate and Stay Resident
       mov     ax,3100h
       mov     dx,(offset Start - Offset ProgramID + 15)/16 + PSP
       int     21h

; Some error happened, don't load and return to DOS

T_HookAlreadyInstalled:
       push    cs
       pop     ds
       mov     ax,0900h
       mov     dx,offset ErrorMessage
       int     21h
       mov     ax,4C01h
       int     21h

ErrorMessage:
       db      "Program already resident, can't reinstall", 0Dh, 0Ah, '$'
```

```
;---------------------------------------------------------------------
; Scan VarArray for Operand ID and IDX, via 'call ScanVarArray'
; ON ENTRY: AL = operand's ID, BX = operand's IDX
; ON EXIT:  CY clear: AL = operand's value (ID uppercase) or
;           AL = operand's complemented value (ID lowercase).
;           CY set: operand's ID or IDX not found
; DESTROYS: AX, SI, Flags
;---------------------------------------------------------------------

ScanVarArray:
       mov     si,offset VarArrayStart
       mov     ah,al                   ; make a copy
       or      al,20h                  ; force uppercase
Scan001:
       cmp     byte ptr cs:[si],al     ; look for operand's ID
       jne     Scan003
       cmp     word ptr cs:[si+1],bx   ; and then for operand's IDX
       jne     Scan003
       mov     al,byte ptr cs:[si+3]   ; AL = data
       test    ah,20h                  ; check if ID was uppercase
       jne     Scan002
       xor     al,0FFh                 ; nope, AL complemented
Scan002:
       clc
       ret
Scan003:
       add     si,4
       cmp     si,offset VarArrayEnd
       jne     Scan001                 ; keep looking....
       stc
       ret                             ; error return

;---------------------------------------------------------------------

       end     Start
```

Anexo

B

BIBLIOGRAFÍA

Referencias Principales

Tomas, R. Blakeslee, Digital Design with Standard MSI & LSI, Wiley, 2e, 1979, pp 169 y ss.

Otras Referencias

Peatman, John B., Digital Hardware Design, McGraw-Hill, 1980 (Algorithmic State Machines, pp. 212 y ss.)
... Microcomputer-Based Design, McGraw-Hill, 1977 (Input/Output, pp. 201 y ss.)

Referencias en Internet

Ladder básico: http://www.plcs.net/contents.shtml

Advantech: http://membres.lycos.fr/mavati/classicladder/

Espresso: http://www.dei.isep.ipp.pt/~acc/bfunc/espres23.zip

Source in tar.gz format: ftp://ic.eecs.berkeley.edu/pub/Espresso/espresso.tar.gz

Documentation/Example in tar.gz format:
ftp://ic.eecs.berkeley.edu/pub/Espresso/espresso-book-examples.tar.gz

http://209.145.252.10/file/c/source/ESPRESSO.ZIP

"ladder languages" gratuitos:
http://www.sourceforge.net/projects/classicladder

Automating Manufacturing Systems with PLCs, Hugh Jack, 2005 (dibujos)
http://claymore.engineer.gvsu.edu/~jackh/books.html