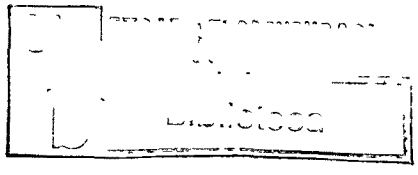
UNIVERSITAT POLITÈCNICA DE BARCELONA

ESCOLA TÈCNICA SUPERIOR
D'ENGINYERS DE TELECOMUNICACIÓ

METODOLOGIA PARA EL DISEÑO GLOBAL DE SISTEMAS BASADOS EN MICROPROCESADOR



TESIS DOCTORAL PRESENTADA A LA UNIVERSIDAD POLITECNICA DE BARCELONA PARA LA OBTENCION DEL GRADO DE DOCTOR INGENIERO DE TELECOMUNICACION

por

MANUEL MEDINA LLINAS

Junio de 1981

DIRIGIDA POR :

ANTONIO ALABAU MUÑOZ

4.3.6.- DESCRIPCION EN LTR.-

La descripcion en lenguaje de transferencia de registros (LTR) precisa a la realizacion de las tareas de mas bajo nivel del sistema, es una de las caracteristicas innovadoras del metodo de diseño propuesto. Ahora vamos a estudiar las ventajas que aporta, las caracteristicas del LTR a emplear y los problemas que se pueden plantear durante la obtencion de la descripcion.

4.3.6.1.- JUSTIFICACION.-

Esta descripcion es la ultima comun a las tareas programadas en LBN y las cableadas. Su objetivo es servir de puente entre estas realizaciones y las descripciones en Pascal-natural o en diagrama de Nassi-Schneidermann. Su existencia esta justificada por varias razones.

* Permite una traduccion casi literal a LBN o a circuito, lo que retrasa la decision final sobre el tipo de realizacion hasta despues de completada esta fase de diseño.

* Simplifica el trabajo necesario para reconvertir una tarea de programa a circuito o viceversa.

* Sirve de documentacion detallada y comentario, para los programas en LBN o los circuitos que se realicen a partir de ella.

4.3.6.2.- CONCEPTO DE LTR.-

Un lenguaje de transferencia de registros, como su nombre indica, es aquel que nos permite describir operaciones de transferencia entre registros. En realidad todos los lenguajes descriptores de circuitos de ordenadores (CHDL) son lenguajes de transferencia de registros. La diferencia entre unos y otros lenguajes indica radica en el nivel de detalle con que permiten describir los circuitos.

Concretando algo mas el concepto de LTR, podriamos situarlo entre los lenguajes descriptores de circuitos (HDL) y los descriptores de ordenadores a nivel de bloques, como el IPS o el PMS. En este sentido se puede incluso decir que los lenguajes ensambladores son unos casos particulares de LTR, en los que parte de las operaciones realizadas por los circuitos del procesador, como incrementar el CP o calcular la direccion efectiva del operando, se suponen implicitamante al especificar una transferencia entre registros y (o) posiciones de memoria o perifericos.

4.3.6.3.- CARACTERISTICAS DEL LTR.-

Al igual que hicimos al hablar del Pascal-natural, y lo mismo que hacen todos los autores que usan LTR como base de sus descripciones, no vamos a proponer aqui una sintaxis concreta para el LTR. En el apendice II se

encuentra una posible definicion de LTR, que es una adaptacion del KARL/CDL a las necesidades impuestas por los objetivos que hemos fijado a la descripcion en LTR.

De una manera general en el LTR debemos distinguir tres componentes basicos: Operadores, transferencias y estructuras de control.

4.3.6.3.1.- OPERACIONES ENTRE DATOS.-

El conjunto de operaciones a emplear en la descripcion debe estar condicionado por las operaciones que tengamos definidas en el nivel inferior. Asi en la descripcion de una tarea en la que se opere con numeros reales y para la realizacion de la cual se va a emplear una UAL de coma flotante, se pueden usar operaciones entre variables formateadas de esta forma. Normalmente, en cambio, deberemos limitarnos al empleo de las operaciones aritmeticas o logicas tradicionales, con operandos formateados en binario puro o complemento a 2, a partir de las cuales se podran definir operadores mas completos.

4.3.6.3.2.- TRANSFERENCIAS.-

Las transferencias de informacion en LTR equivalen a las sentencias de asignacion de LAN. La diferencia entre uno y otro caso es que en LTR tan solo se permiten

expresiones simples, es decir con un maximo de dos operandos y un solo operador. En los LDH se especifica, junto a cada sentencia de transferencia, la condicion en que esa sentencia se ejecutara, pues se supone que todas las sentencias se podrian ejecutar simultaneamente, dentro de las limitaciones de paralelismo en los caminos de datos impuestos por los circuitos. En los LTR, en cambio, se supone la existencia de un secuenciador, con lo que la condicion de ejecucion de una transferencia es implicitamente el final de la ejecucion de la sentencia anterior, salvo los casos de ruptura que veremos mas adelante.

Cuando se quiere recalcar la posible ejecucion en paralelo de varias operaciones, se separan estas por comas, reservando el punto y coma para la separacion entre sentencias que deben ejecutarse forzosamente en secuencia.

4.3.6.3.3.- ESTRUCTURAS DE CONTROL DE FLUJO.-

Existen tan solo sentencias de salto incondicionales y condicionales, pues son las que se pueden realizar mediante circuitos (multiplexores y decodificadores) o mediante instrucciones de lenguaje maquina de un procesador de uso general.

Estas sentencias de salto o condicionales son en realidad transferencias implicitas de constantes al

secuenciador. Las sentencias que pueden seguir a mas de una sentencia iran precedidas de una etiqueta, de esta forma en las sentencias de rotura de secuencia se podra especificar simplemente el nombre de la etiqueta, sin hacer referencia explicita al secuenciador, lo mismo que en lenguaje ensamblador.

Las condiciones que controlan las sentencias condicionales deben ser expresiones logicas simples, es decir relaciones entre un maximo de dos variables simples, o bien referencias a operadores logicos previamente definidos.

4.3.6.4.- DESCOMPOSICION DE LAS EXPRESIONES.-

De todo lo dicho se deduce que la traduccion a LTR plantea basicamente el problema de la descomposicion de las expresiones aritmeticas y logicas en operaciones elementales. Esta descomposicion se puede realizar definiendo un operador compuesto, que caso de programarse se convertiria en una subrutina, y caso de cablearse constituiria un bloque modulo combinatorio. La otra posibilidad de descomposicion consiste en definir unas variables locales intermedias y describir las operaciones simples una tras otra, transfiriendo los resultados parciales a esas variables intermedias.

A nivel LTR la eleccion entre uno u otro procedimiento simplemente significa expresar la

descomposicion dentro de la secuencia de sentencias en que se va a usar, o agrupada junto a otras descomposiciones de operadores fuera de la secuencia de ejecucion. Esta ultima version tiene la ventaja de ser mas modular y por tanto mas util desde el punto de vista documental del sistema. En los niveles inferiores, la equivalencia desaparece, como se vera mas adelante, pero como la eleccion de uno a otro procedimiento no condiciona la realizacion posterior, es recomendable la definicion de operadores por su mayor claridad.

4.3.7.- TRADUCCION A LENGUAJE DE BAJO NIVEL (LBN).-

Una vez limitadas las estructuras de control y definidas todas las variables generales y locales en forma simple o en matriz, para convertir la descripcion en LTR en un programa en LBN, se deben seguir los pasos siguientes:

4.3.7.1.- UBICACION DE LAS VARIABLES.-

Tal como se ha dicho al hablar de la optimizacion de los codigos y de las variables locales, la velocidad de ejecucion del algoritmo y la ocupacion de memoria dependen de la distribucion de las variables entre la memoria central del sistema y los registros internos y acumuladores del procesador. Las variables locales y de almacenamiento temporal de datos intermedios, procuraremos ubicarlas en los algoritmos internos o en zonas de la memoria central facilmente direccionables: pila, pagina, tablas indexadas, etc... De esta forma se podran utilizar instrucciones con campos de direccionamiento mas cortos y ello comportara un acortamiento de los programas y un ahorro de tiempo de ejecucion. Lo mismo podemos decir de las variables mas generadas, con la particularidad de que, al ser su uso menos frecuente, reservaremos las posiciones mas facilmente accesibles a las variables locales.

a)

```
CASO A DE
0,1:    <sentencias 0>
2,3,4: <sentencias 1>
5:      <sentencias 2>
...
```

b)

```
CASO: COMP      A,#0        ; SI A=0
      JUMP,Z    S0          ; ENTONCES ir a S0
      COMP      A,#1        ; SINO SI A=1
      JUMP,NZ   C1          ; ENTONCES
S0:   <sentencias_0>        ; <sent.0>
      JUMP      FC          ; SINO
C1:   COMP      A, #2       ; SI A=2
      JUMP,Z    S1          ; ENTONCES S1
      COMP      A, #3       ; SINO SI A=3
      JUMP,Z    S1          ; ENTONCES S1
      COMP      A,#4        ; SINO SI A=4
      JUMP,NZ   C2          ; ENTONCES
S1:   <SENTENCIAS_1>        ; <sent.1>
      JUMP      FC          ; SINO
C2:   COMP      A,#5        ; SI A=5
      JUMP,NZ   FC          ; ENTONCES
      <sentencias_2>        ; <sent.2>
FC:   ...
```

Fig. 4.12.- Ejemplos de traduccion de la estructura CASO del Pascal-natural a LRI. El lenguaje empleado es el CAIM (Common Assembly Language for Microprocessors), lenguaje comun a todos los microprocesadores, particularizado en este caso para el Z-80.
a) Descripcion en Pascal-natural
b) Codificacion mediante desdoblamiento de la estructura CASO en varias "SI-ENTONCES-SINO". Los comentarios corresponden a una descripcion a nivel LTR.

```
CASO: LOAD     HL, #TABLA
      LOAD     BC,#FC-TABLA+1
      LOAD     IX,#TDS-2
TEST: INC      IX
      INC      IX
      CPIR     TEST          ; SI A<>M(HL) ENTONCES HL:=HL+1; BC:=BC-1;
                             ;             SI BC<>0 ENTONCES salta a TEST
      LOAD     L,(IX)
      LOAD     H,(IX+1)
      JUMP     (HL)
S0:   <sentencias_0>
      JUMP     FC
S1:   <sentencias_1>
      JUMP     FC
S2:   <sentencias_2>
      JUMP     FC
TDS:  DW       S0,S0,S1,S1,S1,S2,FC   ; tabla direcciones sentencias.
TABLA:DB       0,1,2,3,4,5            ; lista de casos.
FC:   ...
```

Fig.4.12.c.- Codificacion mediante consulta de valores en tabla.

4.3.7.2.- ADAPTACION DE LOS MODOS DE DIRECCIONAMIENTO.-

Puesto que en la mayoria de los microprocesadores la otogonalidad de los registros brilla por su ausencia, muchas de las transferencias expresadas en LTR en una sola sentencia, deberan ser desdobladas en dos o mas instrucciones de LBN, utilizando el acumulador como registro puente o inicializando registros indice.

4.3.7.3.- ADAPTACION DE LAS ESTRUCTURAS DE CONTROL.-

La mayor parte de esta tarea ya se ha hecho al escribir las descripciones en diagramas de Nassi-Schneidermann o en LTR, pues y la unica estructura que se ha conservado ha sido la "caso", porque se podia seguir empleando en los LBN.

La traduccion de la sentencia "caso" se puede hacer mediante una tabla de casos posibles, o desdoblandola en tantas sentencias si-entonces como casos posibles haya. El primer procedimiento es el mas indicado cuando el numero de casos posibles es grande, o cuando estan codificados con valores binarios consecutivos, pues el numero de interrogaciones se reduce considerablemente. En la fig. 4.12. se ilustran estos dos procedimientos empleando como LBN el CALM (common assembly languaje for microprocessors) particularizado para el Z80. El CALM es un lenguaje ensamblador propuesto por J.D.Nicoud, que emplea unos nemotecnicos normalizados para todos los

procesadores. En el apendice III se explican los nemotecnicos empleados en el CALM.

4.3.7.4.- CODIFICACION DE LAS EXPRESIONES OPERADORES.-

Como ya se ha dicho al hablar de la traduccion de las sentencias de asignacion a LTR, las expresiones se pueden describir en linea con el resto de operaciones de la tarea, o bien aparte, para lo que se definirian unos operadores especiales. Independientemente de la eleccion hecha en la descripcion en LTR, al traducirlo a LBN tenemos tres opciones:

4.3.7.4.1.- CODIFICACION EN LINEA.-

Consiste en describir cada una de las operaciones simples en que se descompone la expresion, en los momentos en que se deben ejecutar. Es ventajoso si solo se usa el operador desde un punto de la tarea, o cuando se desea optimizar al maximo el tiempo de ejecucion, pues se puede aprovechar en cada caso los estados de los registros que almacenan variables temporales. Tiene el inconveniente de ser poco modular y restar claridad al listado distrayendo la atencion de la estructura general de la tarea.

4.3.7.4.2.- MACROINSTRUCCION.-

Consiste en describir el operador fuera de la secuencia de instrucciones del algoritmo, agrupando todas las instrucciones del operador bajo un mismo identificador, que se podra utilizar como una instruccion mas en la codificacion de la tarea, evitando asi la dilucion de las instrucciones que realizan el algoritmo propiamente dicho. El ensamblador se encargara expandir todas las instrucciones que comprende el operador, cada vez que se haga una referencia a el dentro de la descripcion de la tarea. El resultado final es equivalente al de la descripcion en linea, con la ventaja de la claridad del listado y el inconveniente de no permitir una optimizacion a fondo en el caso de que se use el operador en mas de un punto del programa.

4.3.7.4.3.- SUBRUTINA.-

Difiere del metodo de la macroinstruccion en la forma en que el ensamblador interpreta las referencias al operador desde el programa principal. En el caso de la subrutina las instrucciones son codificadas solamente en el punto del listado en que son descritas, y cuando se hace una referencia a ellas desde la tarea, se rompe la secuencia de ejecucion para saltar a esas instrucciones y ejecutarlas, volviendo finalmente a la instruccion siguiente a la de llamada en el programa principal. Tiene el inconveniente de perder tiempo en los saltos de un lado a otro y la ventaja del ganar espacio de memoria si el operador se usa de una vez.

4.3.7.5.- SINCRONIZACION CON TAREAS DE NIVEL INFERIOR.-

Esta sincronizacion supone dos problemas distintos. El primero es el traspaso de argumentos y el segundo es el disparo y la deteccion del final de la tarea subordinada. Ambos problemas tienen soluciones distintas segun que dicha tarea este tambien programada en LBN o este cableada.

Si la tarea de nivel inferior esta programada, el traspaso de argumentos se puede hacer mediante referencias a variables comunes, fijadas en memoria o registros, o a traves de la pila del procesador. La sincronizacion de inicio y fin se hara mediante una llamada y retorno de subrutina.

Si la tarea subordinada esta cableada, los argumentos se traspasaran entrando y sacando datos a traves de puertas de E/S que se correspondan con registros de datos de la tarea cableada. La sincronizacion de inicio y fin se hara escribiendo y leyendo, respectivamente, en un registro de control y uno de estado de la tarea cableada. Puesto que el control de la tarea cableada no corresponde al procesador central este puede seguir ejecutando el algoritmo de la tarea principal hasta que necesite los datos resultado del tratamiento efectuado por la tarea secundaria. De esta forma aparece

De esta forma aparece el problema del paralelismo, que no vamos a abordar en profundidad, pero que conviene

resaltar. Un ejemplo elemental de paralelismo lo tenemos en la UART, que inicia el proceso de serializacion de un dato en cuanto lo recibe en su registro, sin necesidad de recibir otra orden especifica en el registro de control, y permitiendo al procesador ejecutar las tareas necesarias para preparar el siguiente dato a serializar, que puede ser almacenado en cuanto empieza a ser serializado el ultimo que se escribio en el registro de dato.

Un caso especial de paralelismo se plantea cuando la tarea cableada, esta en realidad controlada por otro procesador. En este caso la sincronizacion y transferencia de argumentos se pueden hacer un nivel mas inteligente, arbitrando mecanismos de Memoria comun, con acceso directo a ella por parte del procesador periferico, ademas del procesador principal. La sincronizacion ahora no se limita a detectar el inicio y el final, sino la modificacion paulatina de los datos por ambos procesadores pues la inteligencia de que hacen gala les permite procesar datos o resultados parciales, antes de tenerlos completos. Para esta sincronizacion suelen emplear semaforos e instrucciones de Lectura y modificacion (test and set), que bloquean el acceso a la zona de datos comun a uno de los dos procesadores, mientras el otro esta modificando esos datos.

4.3.8.- TRADUCCION A CIRCUITOS.-

La traduccion a circuitos de las tareas de mas bajo nivel, descritas en LTR, supone un trabajo completamente distinto del realizado hasta ahora, pues significa pasar de una descripcion secuencial a un esquema logico. Este salto de uno a otro dominio no es nuevo, puesto que ya es empleado en los sistemas de diseño automatico y de simulacion. El esfuerzo realizado hasta ahora para describir las tareas a cablear no es en modo alguno esteril, a pesar de que muchos diseñadores tradicionales comenzarian el diseño del circuito sin haberlo descrito precisamente en forma algoritmica o en LTR. La utilidad del trabajo realizado es doble, de una parte nos ha servido para definir formalmente los parametros de la tarea y las acciones a ejecutar por esta, separando de esta forma el trabajo de analisis del de sintesis, que seguira. De otra parte las descripciones en niveles decrecientes de abstraccion nos proporcionan una documentacion del circuito, que vamos a diseñar, adaptada a las exigencias de cualquier persona interesada.

A continuacion se comentan los aspectos mas importantes de esta fase de diseño, asi como las alternativas existentes en funcion de la complejidad de la tarea a diseñar.

4.3.8.1.- SINCRONIZACION CON LAS TAREAS DE NIVEL SUPERIOR.-

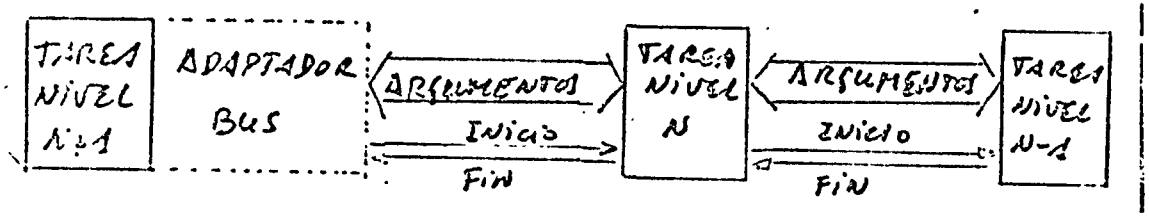


Fig. 4.13.- Relaciones logicas entre tareas. El circuito adaptador de bus solo existira si la tarea de nivel N+1 esta programada y la de nivel N esta cableada. La señal de INICIO indica a la tarea de nivel inferior la disponibilidad de los argumentos de entrada y por tanto el permiso para iniciar su tratamiento. La señal de FIN indica a la tarea de nivel superior la disponibilidad de los argumentos de salida y por tanto el final del tratamiento de los datos.

Desde el punto de vista de una tarea cableada, la sincronizacion con las tareas de nivel superior implica la captacion de ordenes y datos, y la devolucion de resultados e indicaciones de estado. Los datos y los resultados constituyen los argumentos de entrada y salida de la tarea, respectivamente, y se transferira a traves de registros accesibles a ambas tareas, considerando la acepcion registro en el sentido mas amplio imaginable. Las ordenes de control y las indicaciones de estado se pueden considerar tambien como argumentos de la tarea, con la sola diferencia de que su valor sera interpretado de distinta manera, afectando directamente a las condiciones de control del algoritmo ejecutado por cada tarea.

De entre las ordenes de control debemos destacar la de "inicio", que pondra en marcha la tarea cableada, indicandole que tiene a su disposicion una nueva coleccion de argumentos de entrada, que puede comenzar a tratar. De entre los indicadores de estado cabe destacar el de "fin", que señala el final de la ejecucion del algoritmo desarrollado por la tarea cableada, e indicar a la tarea de nivel superior que tiene a su disposicion el juego de resultados obtenidos del tratamiento de los argumentos de entrada. Estas dos señales son equivalentes a las de llamada y retorno a subrutina, cuando la tarea subordinada esta programada en vez de cableada, y constituyen el minimo de informacion que se transmitira entre dos tareas cooperantes.

Para una tarea cableada, la forma de generar o

detectar las señales de sincronismo con las tareas subordinadas o superiores, es independiente del procedimiento de realizacion de estas. De esta forma se evita que cambios en el procedimiento de realizacion de una tarea superior afecten a los circuitos de control diseñados.(fig.4.13). Evidentemente esta equivalencia solo se cumple a nivel logico, pues a nivel electrico, cuando la tarea de nivel superior este programada, se debe transmitir la informacion a traves del Bus del procesador, para lo cual hay que introducir un circuito de adaptacion entre el bus y la tarea cableada. Este adaptador del bus comprendera simplemente los circuitos amplificadores de datos decodificadores de direcciones e interpretadores de las señales de control del bus.

4.3.8.2.- DISCRIMINACION DE LAS PARTES DE CONTROL Y DE TRATAMIENTO DE DATOS.-

Mientras las tareas estaban programadas, se ha supuesto la existencia de un procesador encargado de secuenciar las instrucciones en el orden correcto, ya sea correlativo o no, y una unidad aritmetico-logica encargada de efectuar las modificaciones pertinentes en los datos. Al realizar la tarea mediante un circuito especializado, este debera cumplir las dos funciones y por tanto nos interesara diferenciar la parte del circuito encargada de manipular los datos de la parte encargada de controlar esas manipulaciones.

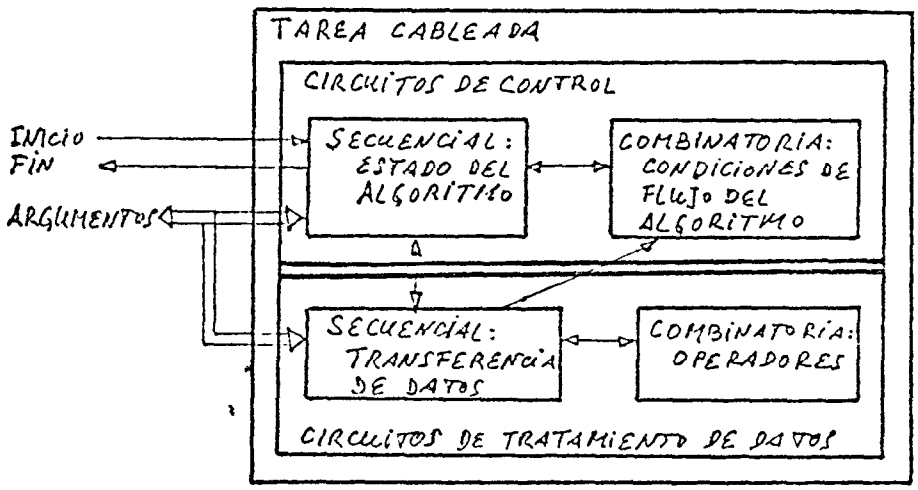


Fig. 4.14.- Bloques generales en los que se puede descomponer toda tarea cableada. Esta descomposicion inicial permite dividir el problema de diseño del circuito en cuatro partes bien definidas y relativamente independientes.

Dentro de los circuitos de transferencia podemos distinguir tambien dos partes: La parte combinatoria que realizara los operadores empleados en la ejecucion del algoritmo, ya sean basicos o definidos especificamente para esta tarea; y la parte secuencial, que se reducira entonces a los biestables y registros en los que se almacenaran los datos (fig.4.14).

Los circuitos de control tambien se pueden considerar divididos en una parte combinatoria y una secuencial. La primera calculara en todo momento las condiciones que controlan la ejecucion del algoritmo; y la segunda memorizara el estado de la tarea en cada instante. Por estado de la tarea se entiende cada una de las operaciones o conjuntos de operaciones que se pueden ejecutar en un instante determinado, asi pues, cada tarea se puede interpretar como una maquina secuencial, cuyo grafo de estados se corresponde biunivocamente con el algoritmo ejecutado por la tarea.

4.3.8.3.- DESCRIPCION EN DIAGRAMA DE BLOQUES DE CIRCUITO.-

Los diagramas de bloques constituyen el metodo mas empleado para describir los circuitos de una forma general, sin entrar en detalles de caracter puramente electrico.

Dentro del concepto "diagrama de bloques" se pueden incluir esquemas con un nivel de abstraccion muy diverso,

desde los absolutamente generales, como el de la fig.4.14, hasta los que tan solo diferiran del esquema electrico definitivo en la realizacion de funciones OR con puertas NAND o en la especificacion de componentes discretos y amplificadores de bus. Segun el tipo de informacion aportada por el diagrama, distinguimos tres clases de diagramas.

4.3.8.3.1.- BLOQUES DE TRATAMIENTO.-

Este primer diagrama es traduccion practicamente literal de la descripcion en LTR. En el se indican tan solo las estructuras de datos, los caminos posibles de estos entre ellas y los bloques operadores interpuestos en esos caminos. La parte de control se reduce a un unico bloque, del que salen las señales necesarias para controlar las transferencias de datos por uno u otro camino y entran las señales que condicionan la secuencia de ordenes de control a generar. Este primer diagrama servira de indice de los siguientes y no presupone la forma en que se vaya a realizar la parte de control, si cableada o microprogramada.

4.3.8.3.2.- BLOQUES DE CONTROL.

Detallan la parte de control de la tarea, ya sea microprogramada, ya cableada. En el primer caso constara tan solo de un contador de programa una memoria de microprogramas y la logica para generar las condiciones de salto condicional, no vamos a insistir en este caso de

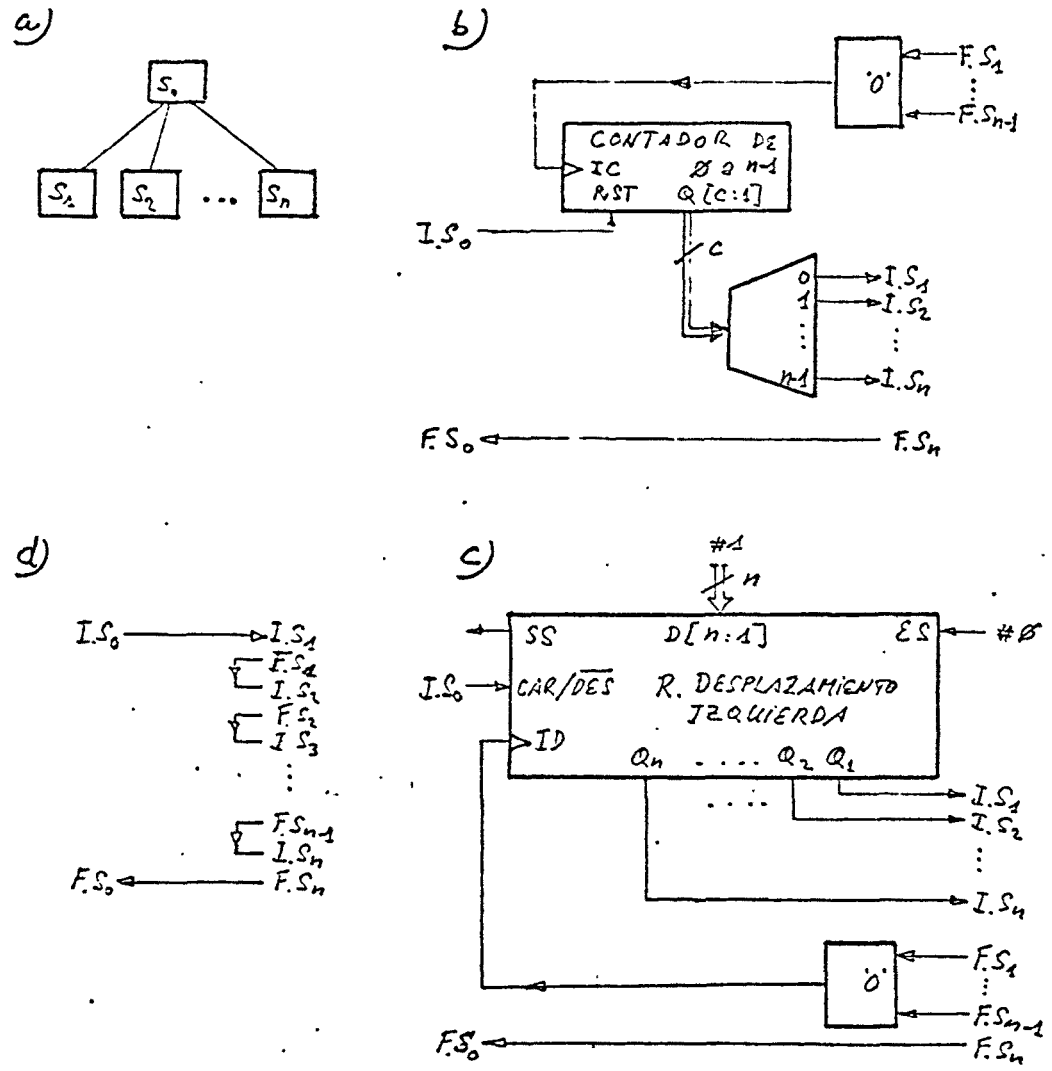


Fig. 4.15.- Estructura de secuencia (a) y diagramas de bloques equivalentes. Los circuitos excitan sucesivamente las señales de inicio de cada una de las sentencias que componen la secuencia. Las señales de fin de cada una de ellas marcan el instante en que se debe iniciar la siguiente sentencia. La suma lógica de las señales de fin de secuencia se hará mediante una puerta OR o mediante circuitos más complejos, dependiendo de la zona activa de la señal (nivel o flanco). Según la forma de codificar el estado interno tenemos tres casos:

b) El estado de la sentencia está codificado en binario en el registro CONTADOR.

c) El estado está codificado en unario (uno entre n) en el registro de desplazamiento.

d) El circuito es completamente pasivo y no almacena información sobre la sentencia Sj, que se está ejecutando.

a)

```
CØ: SI (cond.)
      ENTONCES (sent. sí)
      SINO (sent. no)
CF: . . .

SØ: CASO (exp.) DE
      (valor 1): (sent. 1)
      (valor 2): (sent. 2)
      . . .
      (valor n): (sent. n)
SF: FIN
```

b)

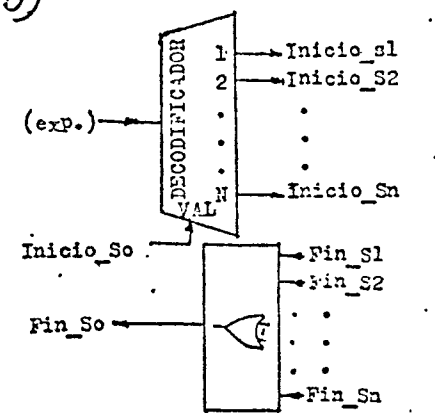


Fig. 4.16.- Estructuras CASO y SI/ENTONCES/SINO (a) y diagrama de bloques equivalente (b). La condicion C, codificada en binario, excita solo una de las señales de inicio, ISj, al llegar la señal de inicio de la sentencia CASO, ISo.

a)

```
RØ: REPETIR
      (sent. bucle)
    HASTA (cond.)
RF: . . .
```

b)

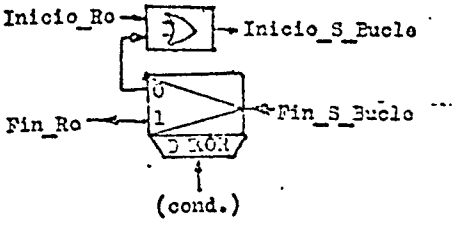


Fig. 4.17.- Estructura de la iteracion: REPETIR/HASTA (a) y diagrama de bloques equivalente (b). La condicion binaria (Si=1; No=0) de final del bucle, C, aplicada al multiplexor, consigue que al final de la sentencia a iterar se repita de nuevo o indique el final de la repeticion.

a)

```
MØ: MIENTRAS (cond.)
      HACER (sent. bucle)
MF: . . .
```

b)

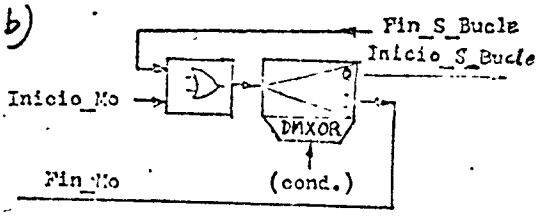


Fig. 4.18.- Estructura de la iteracion: MIENTRAS/HACER (a) y diagrama de bloques equivalente. Difiere del caso de la fig. 4.17, en que la condicion de iteracion encamina la señal de inicio, para que dispare la sentencia a iterar o indique el final de la iteracion.

realizacion, pues existe mucha literatura desarrollada al respecto.

En el caso de que la parte de control este cableada, la resolucion del problema no esta tan sistematizada, y vamos a detallar un poco su descripcion. Hasta ahora hemos seguido un proceso de diseño completamente estructurado y por tanto el algoritmo que debe ejecutar la tarea esta compuesto tan solo de las cuatro estructuras basicas.

Los bloques logicos que realizan estas estructuras deben generar la secuencia de ordenes necesaria para que las sentencias incluidas dentro de la estructura de control se ejecuten en el orden correcto y el numero de veces adecuado. Dado que las sentencias incluidas en una estructura pueden ser a su vez sentencias estructuradas, para explicar una posible descomposicion en bloques funcionales de las estructuras de control del algoritmo vamos a suponer que toda sentencia, simple o compuesta, necesita una señal de "Inicio" para comenzar su ejecucion y genera una señal de "Fin" para indicar el termino de la(s) operacion(es). Con este supuesto, la traduccion a bloques logicos de las estructuras de control del algoritmo puede ser tan simple como se ha mostrado en las figuras 4.15 a 4.18. en las cuales se han empleado bloques combinatorios y secuenciales tipicos, dibujados segun el lenguaje de diagrama de bloques del apendice IV.

El lenguaje propuesto tiene como objetivo conseguir

que los diagramas de bloques sean autoexplicativos, dentro de lo posible. Para ello se fuerza a especificar el sentido en que fluye cada señal, el numero de lineas que componen un camino multiple (bus) y los niveles o flancos activos de las señales.

La informacion contenida en los diagramas de bloques de este nivel es mayor que la expresada por los diagramas de bloques de tratamiento o por las descripciones en LTR. La informacion aportada consiste en combinar todas las condiciones en las que se debe llevar a cabo una transferencia por un camino determinado, seleccionando las posibles fuentes de informacion y los destinos correctos en funcion de las señales de "inicio" de las sentencias correspondientes. Estas combinaciones se realizan todavia a nivel logico, sin especificar la forma en que se combinaran , por ejemplo, dos señales activas en sus flancos.

Una alternativa a este lenguaje digna de consideracion la constituye el ABL, propuesto por HARTENSTEIN como descripcion paralela al CDL/KARL, y que no hemos adoptado directamente por considerarala demasiado proxima a la descripcion en LTR y con una copmplejidad que no hace evidente su lectura, por lo que no cumple el objetivo fijado de servir de puente entre la descripcion LTR y el circuito logico.

4.3.8.3.3.- BLOQUES FUNCIONALES.-

Por bloque funcional se entiende todo circuito de complejidad igual o equivalente a la de los circuitos MSI (o LSI) disponibles en el mercado.

El objetivo de un diagrama de bloques a este nivel es cubrir la ultima etapa previa al diseño del circuito electrico definitivo, centrando la atencion del diseñador en la funcion logica realizada por cada señal y cada bloque, sin necesidad de detallar a nivel de puertas e inversores la forma en que se van a realizar los bloques funcionales. Bloques funcionales pueden ser un multiplexor de dos canales, un decodificador de un bit o un registro de desplazamiento bidireccional con carga serie y paralelo de 32 bits., por citar algunos ejemplos en los que se cumple la condicion expresada al principio, pues son circuitos de complejidad equivalente a la de pastillas reales pero que no se fabrican por problemas de encapsulado o de mercado, o no se pueden emplear por problemas de mantenimiento o economia.

En estos diagramas se deben detallar todos los bloques logicos cuya complejidad no hace inmediata su traduccion a circuitos fisicos. En este proceso se deben conservar todas las señales de control y de datos entre bloques con el mismo significado, dejando las posibles simplifiaciones de inversores, puertas o multiplexados para el esquema logico a cablear.

4.3.8.4.- DESCRIPCION EN LDC.-

Hasta aqui hemos supuesto que la descripcion en LTR, con que contabamos al inicio del proceso de diseño especifico de la tarea cableada, era suficientemente detallada para la confeccion del circuito correspondiente. Esto sera cierto siempre que la tarea no tenga una estructura excesivamente compleja y si la sincronizacion con las tareas programadas esta controlada por estas, mediante instrucciones de E/S. En los casos mas complicados, o cuando la sincronizacion con la tarea programada este controlada por el propio circuito de la tarea (acceso directo a memoria o interrupcion), se debe detallar la descripcion a nivel LTR, incluyendo sentencias y condiciones de secuenciamiento y haciendo referencia a las señales de control del bus y a los registros temporales que sean necesarios.

La descripcion a este nuevo nivel de detalle se puede hacer en LTR si el secuenciamiento es obvio. Si por el contrario el secuenciamiento es complicado, por la propia estructura de la tarea, o porque se pretenda paralelizar el maximo de transferencias en cada ciclo de reloj o porque se quiera renunciar al sincronismo del sistema para conseguir un maximo de velocidad, entonces se debe emplear en la descripcion un lenguaje mas potente, como lo son los LDC, de entre los que destacamos como mas utiles, para el tipo de problemas que se nos van a plantear, el CDL y su variante el KARL, de los que ya hablamos en el cap. 1.

Cuando se considere necesaria una descripcion mas detallada en LTR o LDC, se tomara esta como referencia

para la confeccion de los diagramas de bloques y esquemas logicos de la tarea.

4.4.-RECONVERSION DE PROGRAMA A CIRCUITO.-

Ya se ha dicho que una de las ventajas fundamentales del metodo de diseño descrito es precisamente la facilidad con que se puede realizar esta reconversion. Ahora vamos a analizar las cosas en las que es util llevarla a cabo y los problemas que se pueden presentar.

4.4.1.- UTILIDAD DE LA RECONVERSION.-

Una vez realizadas todas las tareas, se puede verificar si se cumplen o no los supuestos hechos durante el diseño, respecto a la velocidad de ejecucion o la ocupacion de memoria. En el caso de que alguna tarea programada no sea suficientemente rapida para tratar la informacion que le llegue o deba enviar al exterior, sera inprescindible realizar la tarea mediante un circuito especializado.

En otros casos puede suceder que la ocupacion de memoria sobrepase nuestras previsiones, lo cual nos obligaria a aumentar el numero de decodificadores, amplificadores o incluso tarjetas del sistema previstas; en este caso puede resultar conveniente cablear alguna tarea programada, de forma que se ahorre la memoria de datos y programa sobrantes, a cambio de añadir unos circuitos de coste inferior al que supondria el coste de la memoria adicional. Este problema es tipico de las realizaciones en las que se pretende emplear un

procesador monopastilla, pues en estos sistemas la inclusion de memoria externa es costosa, hasta el extremo de obligar a cambiar las instrucciones del programa para hacer referencia a datos contenidos en la memoria externa en vez de la interna, como sucede por ejemplo en los procesadores de la familia 8048.

4.4.2.- CASO DEL CIRCUITO A MEDIDA.-

Cuando por alguna de las razones expuestas se debe reconvertir una tarea de programa a circuito, el metodo de diseño empleado nos permite aprovechar la mayor parte de la labor realizada, pues tan solo es inutil la labor de codificacion de la tarea en LBN.

Para realizar la tarea por programa bastara con tomar su descripcion en LTR y proceder de la misma forma indicada para las tareas cableadas directamente. Ahora, ademas, tenemos la ventaja de que hemos podido verificar el correcto funcionamiento del algoritmo reflejado en la descripcion LTR, pues ya habra sido ejecutado por programa en el procesador y se habran eliminado los errores existentes.

El cableado de una tarea programada implica tambien pequeños retoques en las tareas relacionadas con ella, pues como ya se ha dicho los mecanismos de sincronizacion y traspaso de argumentos son diferentes. Basicamente estas modificaciones se reduciran a sustituir instrucciones de

transferencia con la memoria y de llamada a subrutina, por instrucciones de E/S en las tareas de nivel superior. En las de nivel inferior bastara con suprimir la etapa de adaptacion al bus del procesador, conectando directamente las señales de sincronismo y datos compartidas por ambas tareas.

4.4.3.- CASO DEL CIRCUITO GENERAL PARTICULARIZADO.-

Cuando el volumen o la complejidad de las tareas a cablear sea considerable puede resultar conveniente emplear en su realizacion un procesador especializado, ya sea un procesador de uso general ya un circuito programado por el fabricante para realizar un tipo de tareas especifico. En el primer caso la tarea de conversion consistiria en traducir la descripcion en LTR al ensamblador del nuevo procesador empleado, suponiendo que fuera distinto del principal. En el caso de emplear un circuito especializado o controlador LSI, bastara con adaptar las opciones que ofrezca el circuito a las necesidades concretas de la tarea a realizar. Como ejemplo del primer caso podemos citar el UPI-41, usado por Intel como procesador especializado en tareas de E/S. Para el segundo caso podemos citar como ejemplos la USART o los controladores de teclado y visualizador, de TRC, de disco flexible o de procedimientos de linea, cada vez mas empleados para realizar tareas de control de los perifericos mas normales.

Si el circuito empleado en la realizacion de la tarea es una pastilla de control, la sincronizacion con ella se hara de la misma forma que con un circuito cableado con componentes MSI y SSI, con la sola diferencia de que se deberan adaptar los registros de control previstos a los realmente existentes en la pastilla.

Si se emplea un procesador de uso general en la realizacion de la tarea o tareas reconvertidas a "circuito", la sincronizacion con las tareas de nivel superior, programadas en el procesador central, se puede hacer de la misma forma que entre cualquier par de tareas programada y cableada, con la unica particularidad de que los registros de datos y control de la tarea "cableada" seran en realidad posiciones de memoria o puertas de entrada del procesador huesped (especializado). Pero tambien se pueden emplear mecanismos de transferencia de argumentos y sincronizacion mas sofisticados, consistentes en compartir una parte de la memoria del procesador principal (hostelero), lo cual permite un grado de paralelismo en la ejecucion de las tareas mucho mayor, aunque plantea el inconveniente de una sincronizacion mas compleja.

4.4.4.- PARALELISMO EN LA EJECUCION DE LAS TAREAS.-

Para conseguir el maximo de paralelismo, sin complicar el mecanismo de sincronizacion, basta con modificar el orden de las operaciones en las tareas de

nivel superior, de forma que se disponga de todos los parametros actuales lo antes posible, para transferirlos y disparar la ejecucion de la tarea de nivel inferior en cuanto se tengan. A partir de ese momento se deben realizar todas las sentencias que no dependan de los resultados a obtener por la tarea de nivel inferior, de forma que al final no se tenga que esperar para obtener los resultados, o la espera quede reducida al minimo inprescindible. En este caso el mecanismo de sincronizacion se reduce al ya mencionado, con dos unicas primitivas de "Inicio y Fin".

Complicando el mecanismo de sincronizacion, se pueden dividir los argumentos de entrada y salida de la tarea en grupos, con lo que la tarea de nivel inferior puede iniciar el tratamiento de los datos mientras la de nivel superior todavia esta calculando algunos de los parametros que debe transferir, por lo que tal vez el procesador hospedado debera esperarlos durante la ejecucion de la tarea subordinada. Analogamente el procesador hostelero tal vez debera esperar para obtener los ultimos resultados, despues de haber obtenido y procesado los primeros. En el limite habria un indicador para cada argumento. El problema de la sincronizacion se puede complicar aun mas si se extrapola el caso y se piensa en la posibilidad de que haya varios procesadores hospedados, ejecutando tareas utilizables por mas de un procesador.

4.5.- RECONVERSION DE CIRCUITO A PROGRAMA.-

Es el caso inverso al anterior y presenta analogas ventajas. Vamos a analizar las condiciones en las que conviene realizar la reconversion y los problemas planteados por un caso tipico.

4.5.1.- UTILIDAD DE LA RECONVERSION.-

En el apartado anterior comentabamos la posibilidad de que faltase memoria o tiempo para ejecutar todas las tareas programadas inicialmente, pero tambien puede suceder lo contrario, es decir, que sobren algunas posiciones de memoria y tiempo del procesador. En este caso tal vez se puedan ahorrar componentes, consumo y espacio, programando alguna de las tareas inicialmente cableadas. En general esta situacion no sera tan frecuente como la de traducir programas a circuitos, pues al calcular la situacion inicial de la barrera programa/circuito, ya dijimos que se debia situar en el punto mas bajo posible del arbol de tareas del sistema. No obstante se puede pensar, por ejemplo, en programar tareas realizables mediante circuitos especializados economicos, como la UART, pues la sencillez de la realizacion por circuito nos puede haber inducido a colocar inicialmente la barrera P/C sobre la tarea de serializacion.

4.5.2.- APROVECHAMIENTO DE LA POTENCIA SOBRANTE DE UN PROCESADOR SECUNDARIO.-

Ya hemos visto que la complejidad de alguna de las tareas a cablear, o el conjunto de ellas, puede aconsejarnos emplear un procesador adicional para realizarlas en vez de cablearlas con componentes directos. En ese caso, aparece una barrera P/C adicional, para el nuevo procesador, cuya colocacion debemos considerar de forma analoga a la descrita para el procesador principal. De esta consideracion resultara, en la mayoria de los casos, que el nuevo procesador cuenta con capacidad de calculo y memoria de sobra y, por tanto, algunas de las tareas, que no se podian programar en el procesador principal por problemas de saturacion y que no fueron suficientemente complejas como para aconsejar el uso de un procesador adicional, ahora podran ser programadas sobre el nuevo procesador, reconvirtiendolas asi de circuito a programa.

En este caso se ha supuesto que los procesadores que componen el sistema global estan jerarquizados, como consecuencia del proceso descendente empleado en el diseño, que habra ido "saturando" los procesadores con tareas cada vez mas secundarias. Esta suposicion descarta la posibilidad de diseñar arquitecturas multiprocesadores con asignacion dinamica de tareas, pues su empleo requiere los servicios de un programa supervisor, que normalmente se diseñara a parte, y que nos permitira considerar el conjunto de procesadores cooperantes como uno solo, con

una capacidad equivalente multiplo de la uno de los procesadores fisicos.

Como conclusion de todo lo visto podemos afirmar que la nueva metodologia nos proporciona una descripcion general y en un solo lenguaje, tanto de las estructuras de los datos que se manejan, como de los algoritmos que ejecutara el sistema, independientemente de que correspondan a tareas programadas o cableadas.

La organizacion jerarquizada de las tareas aporta unos criterios objetivos, facilmente aplicables, para determinar "a priori" el procedimiento idoneo de realizacion de cada tarea. No obstante, la practica ausencia de bifurcaciones importantes en el camino a seguir durante el diseño, hace que esta decision pierda importancia, pues, como se demuestra al final del capitulo, el trabajo de reconversion de un procedimiento de realizacion a otro es puramente mecanico y aprovecha practicamente todo el trabajo desarrollado para obtener la primera realizacion. Esta facilidad permitira, en muchos casos, comprobar los algoritmos simulandolos en lenguaje de alto nivel o ensamblador, antes de alcanzar los ultimos tramos del proceso de diseño, facilitandose de esta forma la deteccion de los errores de concepto, y dejando para la realizacion final, tan solo posibles errores de transcripcion de un lenguaje a otro.

Finalmente se debe resaltar la sistematizacion de la traduccion de descripciones secuenciales de algoritmos a diagramas de bloques de circuitos, pues esta fase del

diseño es esencial para poder aprovechar todo el trabajo de analisis y descripcion de las tareas a programar, y ademas constituye una verdadera novedad, aun no contemplada por los metodos de diseño de circuitos logicos convencionales.

5.- PUESTA A PUNTO DE SISTEMAS DISEÑADOS CON EL METODO BARNA.-

Una de las caracteristicas basicas del metodo de diseño propuesto hasta ahora es la utilizacion de un mismo procedimiento para el analisis y la concepcion de todas las tareas que componen el sistema, independientemente de la forma de realizacion final. Esta caracteristica se mantiene tambien en la fase de puesta a punto, pues al basarse esta en la documentacion de las tareas y ser esta analoga para todas ellas, el procedimiento de verificacion de las tareas sera tambien comun para todas, con independencia del procedimiento de realizacion, salvo, claro esta, las herramientas empleadas en la comprobacion, pues dependen de la forma y la situacion, de las señales a analizar.

En este capitulo se describe el metodo de puesta a punto a seguir para aprovechar al maximo la documentacion generada en la fase de sintesis de las tareas, con el fin de simplificar y sistematizar al maximo el trabajo de verificacion.

5.1.- CONSIDERACIONES GENERALES.-

En el capitulo 3 se ha hablado de las herramientas desarrolladas para la puesta a punto de sistemas en los ultimos años. Ahora vamos a seleccionar aquellas que consideramos mas adecuadas al tipo de problemas con que habitualmente se va a encontrar el diseñador de sistemas basados en microcomputador.

5.1.1.- PROCEDIMIENTOS DE VALIDACION.-

Vamos a considerar dos aspectos de la validacion, que podriamos denominar la forma y el fondo. En cuanto a la forma se estudiaran las dos alternativas: Ascendente y Descendente. En cuanto al fondo analizaremos las posibilidades de realizacion de los metodos selectivo y exhaustivo.

5.1.1.1.- ORDEN DE LA VALIDACION.-

Existen dos formas posibles, la ascendente y la descendente, con un numero similar de defensores para cada una de ellas. Puesto que el objetivo de nuestro trabajo es la puesta a punto de un sistema descompuesto en tareas programadas y en tareas cableadas, uno de los problemas a resolver consiste precisamente en la verificacion de que las tareas programadas se pueden ejecutar en el tiempo disponible y que las tareas

cableadas estan bien sincronizadas con las tareas programadas. Por esta razon consideramos conveniente comenzar la verificacion con las tareas mas proximas a la barrera programa/circuito, de esta forma esos posibles asincronismos, o inadecuaciones entre los objetivos de la tarea y el medio de sintesis elegido para ella, se haran evidentes desde las primeras etapas de la verificacion.

En consecuencia se aplicara la forma descendente en la verificacion de las tareas cableadas y la forma ascendente para las tareas programadas. En el proceso de puesta a punto descendente se sustituiran las tareas de nivel inferior, aun no realizadas, por tareas vanales, que simplemente permitan proseguir la ejecucion de la tarea de nivel superior. En la puesta a punto ascendente no existe este problema, pues las tareas de nivel inferior ya han sido realizadas, en cambio existe el problema de la inicializacion de la tarea a verificar, pues para ejecutarla hara falta transferirle unos argumentos, lo cual se debera hacer desde unas rutrinas de comprobacion, cuya constitucion analizaremos mas adelante.

5.1.1.2.- PROFUNDIDAD DE LA VALIDACION .-

Considerando que en la mayoria de los sistemas basados en microprocesadores, el coste de los programas puede suponer hasta diez veces el coste de los circuitos de un equipo, sobre todo en los sistemas mono- y bi-pastilla. Esta economia relativa de los circuitos hace que

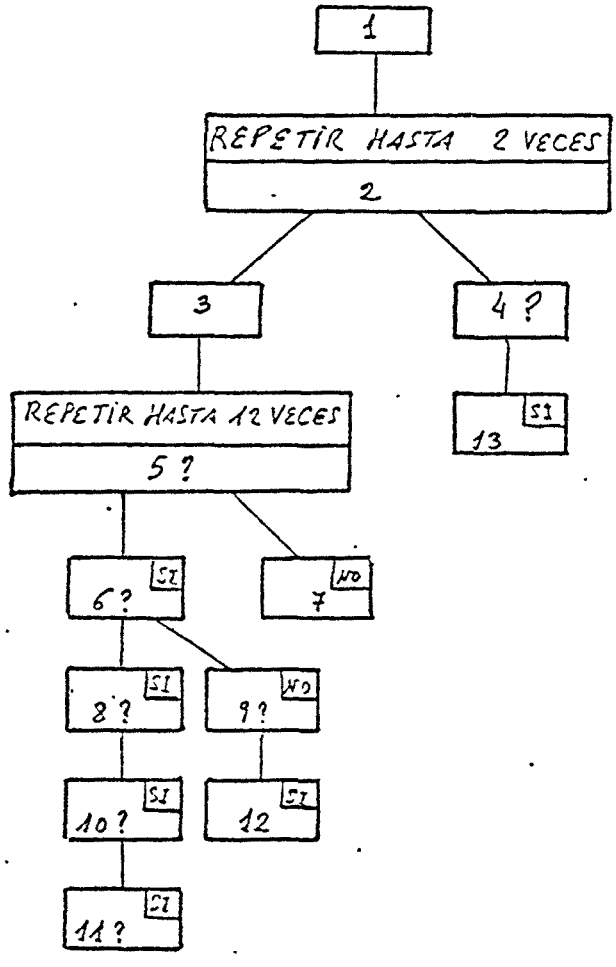


Fig. 5.1.- Diagrama de Jackson de una tarea simple. Los bucles y las alternativas de ejecucion hacen que existan del orden de 1,9 x 10^19 caminos posibles distintos entre el principio y el final.

los costes de la programacion se hagan mas patentes. De estos costes, mas del 50% corresponde a la verificacion de los programas, pudiendo variar en mas o menos el 20% dependiendo del metodo empleado.

La utilizacion de la programacion estructurada en la concepcion de los programas simplifica mucho las tareas de verificacion, pues da unas ciertas garantias de finitud y correccion estructural al programa producido.

Ello hace que no sea necesario realizar la comprobacion de la tarea con un conjunto exhaustivo de datos, sino que se puede ejecutar simplemente con un conjunto de valores iniciales que garantice la ejecucion de todas las ramas de la tarea al menos una vez.

Mas adelante analizaremos la forma en que se determinan exactamente los bloques de la tarea a verificar y las condiciones necesarias para conseguirlo, pero de momento vamos simplemente a recalcar que un programa simple, como el de la fig. 5.1., puede ejecutarse siguiendo del orden de 1.9 x 10^19 caminos posibles, segun se cumplan o no cada una de las 8 condiciones que plantea su estructura, lo cual deja bien clara la imposibilidad de comprobar todos los caminos posibles de ejecucion de un programa.

5.1.2.- APROVECHAMIENTO DE LA DOCUMENTACION.-

La determinacion de los segmentos de algoritmo a verificar y de los puntos significativos a trazar se hace a partir de la descripcion general de la tarea, que tenemos en forma de diagrama de Jackson. La descripcion a este nivel es la ideal para determinar los posibles caminos logicos a seguir en la ejecucion de la tarea, independientemente de los caminos ficticios que hayan podido aparecer durante el diseño, como consecuencia de limitaciones de los lenguajes de diseño empleados. Ademas, el procedimiento de diseño seguido tiene la ventaja de ofrecernos un mismo tipo de descripcion general para todas las tareas, independientemente de su complejidad final, o del metodo de sintesis empleado, circuito o programa, lo cual nos permite aplicar los mismos criterios de analisis a todas las tareas, para determinacion de las caracteristicas a verificar.

5.1.3.- SELECCION DE PUNTOS DE COMPROBACION.-

Para verificar el correcto funcionamiento de una tarea debemos comprobar que los datos producidos al final de la misma y los enviados a otras tareas corresponden a los que se han introducido al principio y/o los que se han obtenido del exterior. Como en casi todas las comprobaciones, esta verificacion terminal de los datos es valida tan solo para demostrar que la tarea funciona en los casos verificados, pero no permite detectar errores que se hayan compensado por azar.

Ya hemos visto que ejecutar la tarea siguiendo todos los posibles caminos de principio a fin es materialmente imposible, pero lo que si se puede hacer es verificar todos los caminos de cada uno de los bloques en que se descompone la tarea. Los diagramas de Jackson son extremadamente utiles en la determinacion de los bloques de una tarea, pues son tantos como aparecen en el diagrama. Volviendo al ejemplo de la fig. 5.1., podemos apreciar que la tarea cuenta con 13 bloques, cada uno de los cuales puede ser verificado independientemente, considerando los bloques que tiene por debajo como un unico camino, aunque tengan en realidad mas de una opcion, pues estas se verifican con el bloque correspondiente. Al verificar el bloque 6 se comprobaran los dos caminos alternativos 8 y 9, sin preocuparnos las tres opciones que hay en 8 o las dos de 9. Al verificar 3 se comprobara la correcta repeticion de 5, y al verificar 2 se comprobara la correcta sucesion de los bloques 3 y 4, como unico camino posible.

Para evitar realizar ejecuciones fraccionales de la tarea, se colocaran sentencias de comprobacion al principio y al final de cada uno de los bloques a verificar, unificando las comprobaciones que sea posible para evitar engrosar excesivamente la tarea. En estas sentencias de comprobacion se verificaran las aserciones de los puntos correspondientes. Por asercion de un punto se entiende la relacion que debe existir entre las variables de la tarea, todas y cada una de las veces que la ejecucion de la tarea pase por ese punto.

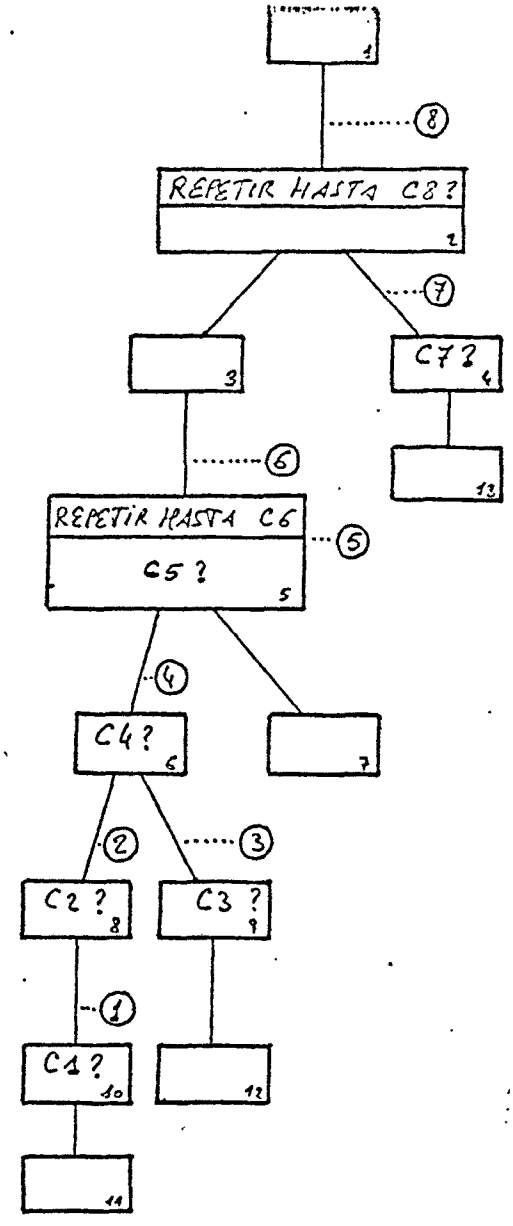


Fig. 5.2.- Diagrama mostrando los puntos de corrretacion. Estos deben colocarse despues de cada una de las sentencias que incluyen la pregunta de una condicion.

Como resumen podemos decir que debemos colocar al menos una sentencia de comprobacion despues de cada sentencia que incluya una condicion (bucle, selectiva, condicional) de ejecucion. Asi en el ejemplo que nos ocupa habria 8 sentencias de comprobacion, tantas como condiciones tiene el algoritmo, y se colocarian en los puntos marcados en la fig. 5.2. mediante unos circulos.

5.1.4.- ELECCION DE LOS UTILES DE TRABAJO.-

En el capitulo 3 se han presentado todos los utiles de trabajo disponibles para la puesta a punto de programas y circuitos simultaneamente, vamos a adoptar como norma general el uso de un emulador en circuito, que tiene la ventaja de permitir la puesta a punto de programas y circuitos, y al mismo tiempo permite verificar el correcto sincronismo entre ambas partes.

El unico inconveniente se presenta cuando las tareas cableadas trabajan con un reloj mas rapido que el del procesador principal, en cuyo caso el emulador en circuito no permite seguir la ejecucion de la tarea cableada, y por tanto no permite su verificacion exhaustiva, sino tan solo en los puntos de sincronismo con las tareas programadas. Para los casos en que esta verificacion rudimentaria no es suficiente, es necesario recurrir a un analizador logico, que permite el uso de un reloj independiente del procesador, y por tanto mas rapido, y que tambien permite analizar simultaneamente

las tareas cableadas y programadas, aunque para estas ultimas sea necesario decodificar los codigos de operacion y las direcciones de memoria, salvo sofisticadas excepciones.

5.2.- DETERMINACION DE LAS CARACTERISTICAS DE LA TAREA A VERIFICAR.-

Las caracteristicas que vamos a determinar son aquellas que nos permitiran verificar que la tarea no funcione mal en los casos iniciales para los que la ejecutemos.

5.2.1.- DETERMINACION DE LOS CAMINOS A EJECUTAR.-

Ya hemos visto que el procedimiento de verificacion que vamos a emplear consiste en ejecutar todas las ramas de la tarea, lo cual permite garantizar que se han verificado todos los bloques en que se descompone la tarea, segun la descripcion de Jackson.

Cada bloque condicional o selectivo abre dos o mas posibles caminos, aunque en algunos casos tan solo uno de esos caminos implique la ejecucion de una accion y por tanto tenga un bloque asociado, como sucede en los bloques 4, 8, 9 y 10 de la fig.5.1. Para calcular todos los caminos posibles debemos añadir esas ramas vacias, tal como muestra la fig. 5.3. En ella se puede observar que hay tantos caminos posibles como bloques terminales del diagrama (incluyendo los bloques vacios). Esto no es siempre cierto, pues cuando una rama termina con una secuencia de bloques, todos ellos pertenecen a un solo camino, pero como veremos esto no tiene importancia en la determinacion de las condiciones necesarias para que se

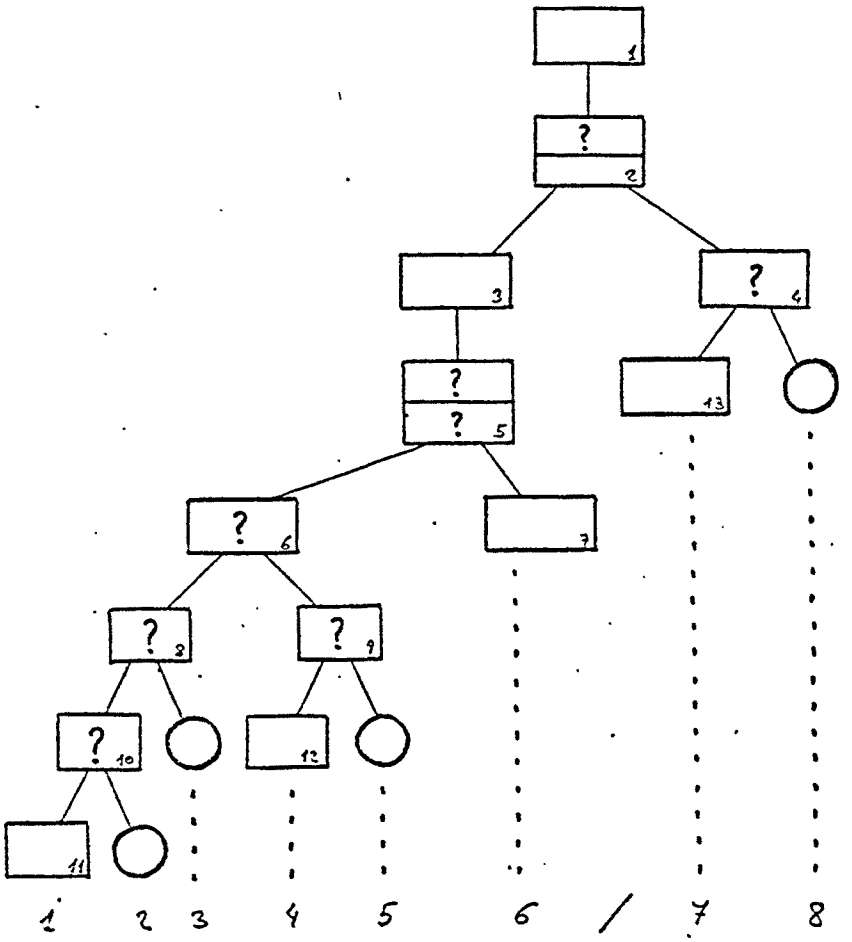


Fig. 5.3.- Diagrama de Jackson correspondiente a la misma tarea de la fig. 5.1., mostrando los bloques opcionales que no implican ninguna accion (vacios). Obsérvese que al haber tan solo un bloque secuencial, al principio de la tarea, solamente hay dos conjuntos de caminos excluyentes entre si. En toda ejecucion se pasara al menos por uno de los 6 primeros caminos y por uno de los dos ultimos (7,8).

ejecuten los caminos. Volviendo al ejemplo de la fig. 5.3., el numero de caminos de la tarea es 8, divididos en dos grupos sucesivos, el primero de 6 alternativas excluyentes, y el segundo de 2.

En la determinacion del numero de caminos a ejecutar no se han considerado los bucles como alternativas, pues se ha supuesto que siempre se debera verificar la tarea pasando al menos dos veces por la condicion de final del bucle, de forma que la primera vez no se cumpla y la ultima si se cumpla, verificando de esta forma todas las opciones con una sola ejecucion, si no consideramos el numero de vueltas dentro del bucle como una alternativa mas a verificar.

5.2.2.- DETERMINACION DE LAS ASERCIONES.-

Ya se ha dicho que una asercion era una condicion que debian cumplir las variables del algoritmo y que permaneceria invariante en todas las ejecuciones del mismo. Otros autores las definen como los sectores de estado del algoritmo.

En toda tarea, procedimiento o sentencia, existen unas variables de entrada y unas de salida, que deben cumplir unos ciertos requisitos antes y despues de ejecutarse la accion correspondiente. Estos requisitos son las aserciones correspondientes a los puntos anterior y posterior a la accion, y reciben los nombres de pre-(p) y

post-condicion (q), tal como se indica en la fig. 5.4.a.

Al definir una tarea, se habran definido las condiciones iniciales de las variables de entrada y las caracteristicas finales de las variables de salida, esto es las pre- y post-condiciones de la tarea. Ya hemos visto que estas caracteristicas de las variables no eran suficientes para verificar el correcto funcionamiento de la tarea y que hacia falta comprobar la correccion de las variables en los puntos de comprobacion mas adecuados (5.1.3). Con lo dicho hasta ahora podemos afirmar que para verificar la tarea debemos comprobar las post-condiciones de las sentencias que incluyen alguna pregunta. (fig. 5.2).

Para calcular las aserciones intermedias de la tarea basta con aplicar las transformaciones realizadas sobre las variables, en cada sentencia, a las variables que intervienen en la pre-condicion para obtener la post-condicion, a la que se deberan añadir las condiciones impuestas para la ejecucion del algoritmo. En la fig. 5.4. se pueden observar las relaciones que aparecen entre pre-y post-condiciones segun el tipo de sentencia ejecutado. En las sentencias de asignacion, la post-condicion se obtiene haciendo que la precondicion quede expresada mediante la misma funcion a evaluar en la sentencia, y entonces basta hacer la sustitucion llevada a termino por la sentencia para obtener la post-condicion. (fig.5.4.b). En una sentencia selectiva, la post-condicion sera la suma logica de las post-condiciones

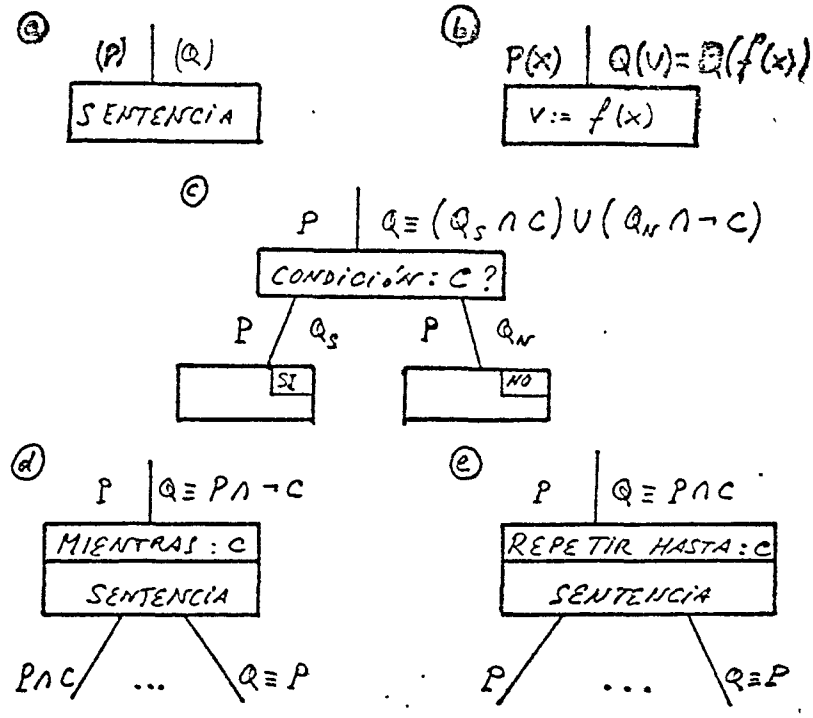


Fig. 5.4.- Diagramas de Jackson mostrando las relaciones entre las pre- y post-condiciones de cada una de las estructuras de control del lenguaje:

a) Forma generica.
b) Asignacion.
c) Condicional o selectiva.
d) Iteracion: Mientras/Hacer.
e) Iteracion: Repetir/Hasta.

alternativas, multiplicada logicamente por la condicion que se debe cumplir para que se haya ejecutado esa alternativa; en general la expresion logica obtenida se podra simplificar (fig.5.4.e). En las sentencias iterativas cabe recalcar que la precondicion de la primera sentencia del bucle debe ser igual a la post-condicion de la ultima sentencia, pues caso de repetirse las sentencias a la ultima seguira la primera; la post-condicion de la sentencia de iteracion sera la pre-condicion multiplicada logicamente por la condicion de final de la iteracion. (fig.5.4.d y 5.4.e).

Como ejemplo del calculo de las aserciones, en la fig.5.5. se ha desarrollado el algoritmo para el calculo del maximo comun divisor de los numeros a y b: mcd(a,b). El algoritmo se basa en las relaciones:

$$mcd\ (a\ b) = mcd\ (b.a.mod.b)$$

$$mcd\ (a\ 0) = a$$

Es interesante recalcar las simplificaciones realizadas en las post- condiciones de las sentencias iterativas y condicional, que son precisamente las que se deberan comprobar en la puesta a punto de la tarea.

5.2.3.- DETERMINACION DE LOS PUNTOS Y VARIABLES A TRAZAR.-

Ya se ha dicho que, a fin de simplificar al maximo la tarea de verificacion, se comprobaria el correcto

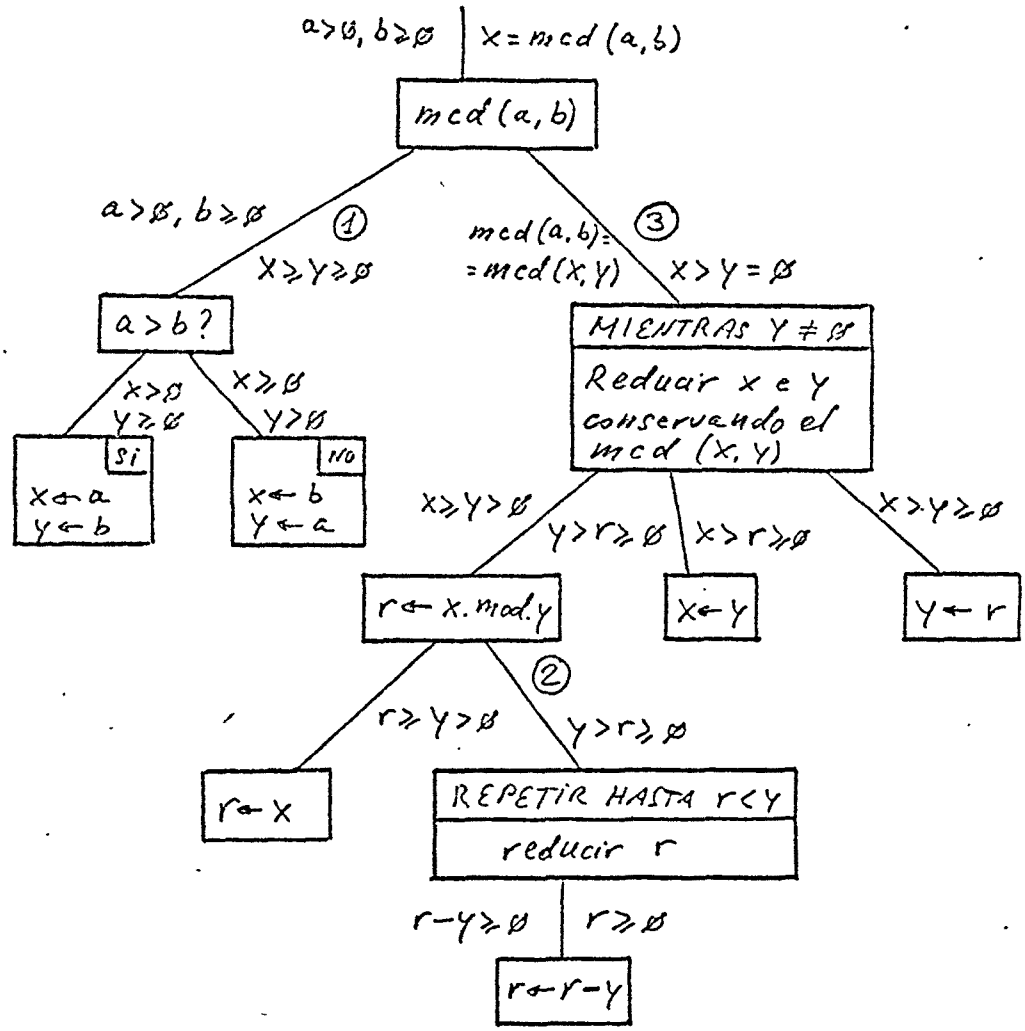


Fig. 5.5.- Ejemplo de diagrama de Jackson para el calculo del maximo comun divisor, incluyendo las aserciones de cada sentencia. Se han subrayado las post-condiciones que se deben comprobar en la puesta a punto, es decir las aserciones correspondientes a los puntos de comprobacion de la tarea, tras haber sido simplificadas:

1: $((X>0, Y>=0) \cap (X>Y)) \cup ((X>=0, Y>0) \cap (X>=Y)) \equiv$
$X>Y>=0 \cup X>=Y>0 \supset X>=Y>=0$

2: $R>=0 \cap R<Y \equiv Y>R>=0$

3: $X>Y>=0 \cap Y=0 \equiv X>Y=0$

funcionamiento de la tarea tan solo en los puntos que siguen a una sentencia que incluye una condicion. Tambien se ha dicho que las aserciones son condiciones suficientes para obtener los resultados deseados, por tanto bastara con verificar que se cumplen las aserciones en los puntos de comprobacion, para lo cual sera necesario trazar las variables que intervienen en dichas aserciones.

Por trazar una variable se entiende obtener la secuencia de valores que va obteniendo dicha variable durante la ejecucion del programa. Como ya veremos mas adelante, esta traza se obtendra intercalando sentencias de comprobacion en el programa que realiza la tarea, o bien programando adecuadamente el emulador en circuito o el analizador logico para que almacene los valores que van tomando las variables durante la ejecucion de la tarea por el circuito.

En el ejemplo de la fig. 5.5. se obtendria la traza de los puntos marcados con un circulo, en los cuales se insertaria una sentencia de impresion de las variables x e y en los puntos 1 y 3, y r e y en el punto 2. De esta forma se podria verificar el cumplimiento de las aserciones subrayadas al termino de las sentencias de control. Es interesante destacar que la sentencia intercalada en el punto 2 se repetira un numero indefinido de veces, pues se encuentra dentro del bucle en el que se va reduciendo "y" hasta que se anule. Por esta razon deberemos estudiar las condiciones de ejecucion de forma que no se repita demasiado el bucle.

En el caso de que la tarea de obtener el maximo comun divisor se realizase por circuito, se deberian almacenar en la memoria de la traza de la tarea los valores almacenados en los registros x, y, r, al finalizar las sentencias de control correspondientes, aprovechando para ello como señal de reloj las señales de fin de sentencia generadas por el circuito en los puntos de comprobacion.

Caso de utilizar un emulador para comprobar la tarea programada, seria necesario trazar las instrucciones en las que se modifican las variables que afecten a las aserciones. Si es posible introducir las sentencias de comprobacion no hay mas que trazar estas sentencias, pero si se quiere evitar la introducion de sentencias que alteren la duracion de la ejecucion de la tarea, se deberan trazar las sentencias en las cuales se actualizan las variables, encontrando el denominador comun (suma logica) de todas ellas para confeccionar la condicion de la traza del emulador, evitando al maximo trazar sentencias o transferencias de dato no significativos.

5.2.4.- SELECCION DEL CONJUNTO DE CONDICIONES INICIALES.-

Ya hemos visto que para verificar una tarea debiamos conseguir que se ejecutasen todos los caminos posibles de cada bloque, al menos una vez. Para ello sera necesario ejecutar la tarea tantas veces como caminos alternativos haya en el tramo del algoritmo mas ramificado. En el caso de que el algoritmo se ramifique y concentre

EJECUCION	CAMINO 1	2	3	4	5	6	7	8	CONDICIONES INICIALES
T1	X						X		$C1 \cap C2 \cap C4 \cap C5 \cap C7$
T2		X						X	$\overline{C1} \cap C2 \cap C4 \cap C5 \cap \overline{C7}$
T3			X				X		$\overline{C2} \cap C4 \cap C5 \cap \overline{C7}$
T4				X				X	$C3 \cap \overline{C4} \cap C5 \cap \overline{C7}$
T5					X		X		$\overline{C3} \cap \overline{C4} \cap C5 \cap C7$
T6						X		X	$\overline{C5} \cap \overline{C7}$

Tab. 5.1.- Tabla de verificacion correspondiente a los diagramas de las fig.5.1, 5.2 y 5.3. En cada ejecucion de verificacion, T1 a T6, se sigue una de las ramas posibles de cada alternativa del algoritmo, precisamente la marcada con una X en la tabla. Como en este caso solo se suceden dos alternativas, cada ejecucion solo tiene marcadas dos ramas. El emparejamiento de estas esta limitado en la practica, por la compatibilidad entre las condiciones iniciales necesarias para seguirlas.

sucesivamente, convendra planificar las condiciones iniciales de forma que en cada ejecucion se siga un camino distinto en todas las ramificaciones, repitiendo alternativas en alguna de ellas tan solo despues de haberlas ejecutado todas o cuando sean incompatibles las condiciones iniciales necesarias para seguir en la misma ejecucion alternativas pendientes en todas las ramificaciones.

Para conseguirlo se confecciona una "Tabla de Verificacion", la cual tiene una columna para cada uno de los caminos del algoritmo, y tantas filas como ejecuciones de comprobacion o verificaiones sean necesarias, indicando en una columna adicional las condiciones iniciales que permiten seguir los caminos marcados para esa verificacion.

En la tabla 5.1. se muestra la tabla de verificacion correspondiente al ejemplo de las figs.5.1, 5.2 y 5.3. Al agrupar los caminos de las distintas ramificaciones en una sola ejecucion hay que verificar que las condiciones impuestas por cada uno de los caminos escogidos son compatibles. En la tabla 5.1 se ha supuesto, por ejemplo, que C4 y C5 eran independientes de C7, que C1 y NO.C3 estaban incluidas en C7 y por tanto debian tomar el mismo valor que esta.

Respecto a los bucles, ya se ha dicho que tan solo debemos garantizar que en alguna de las verificaciones se seguira al menos una vez el camino de retorno, es decir

que las sentencias del bucle se ejecutaran dos veces por lo menos, lo cual en algunos casos nos puede permitir reunir dos verificaciones en una sola ejecucion de la tarea. En la tabla de verificacion no se ha hecho constar esta eventualidad, porque requiere que en la primera ejecucion del bucle se modifique alguna de las condiciones iniciales que determinan la rama del bucle a ejecutaren la segunda pasada, lo cual no siempre es controlable a priori.

La determinacion de la ejecucion de verificacion que repetira un determinado bucle depende de la compatibilidad entre las condiciones respectivas. En los casos en que el bucle esta controlado por un contador o una condicion que no podemos controlar inicialmente, conviene modificar dicha condicion para evitar que durante todas las ejecuciones de verificacion se deba repetir el bucle un elevado numero de veces. Otra opcion, para evitar los bucles no controlables desde el inicio de la tarea, es considerarlos como una subtarea independientemente, lo cual nos permitira fijar las condiciones justo antes del inicio del bucle, salvando los impedimentos existentes entre el inicio de la tarea y del bucle. Esta practica permite ademas considerar el bucle como un solo bloque dentro de la tarea, y por tanto un solo camino, lo cual puede reducir el numero de ejecuciones de la tarea completa.

En la fig. 5.6. se ha desarrollado un ejemplo en el que se puede observar como la eleccion adecuada de las

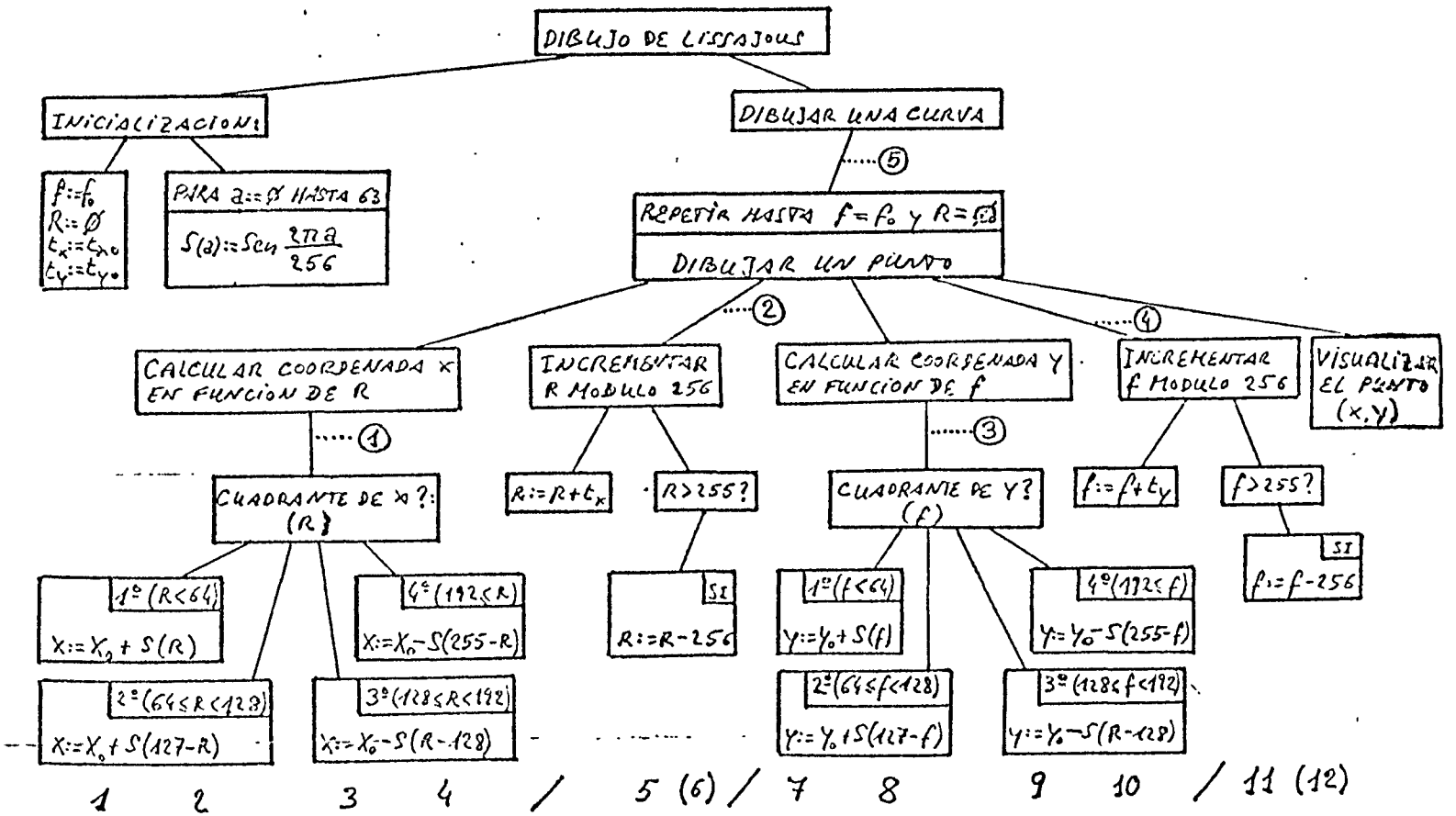


Fig. 5.6.- Diagrama de Jackson de un algoritmo para el dibujo de curvas de Lissajous.

NOTAS aclaratorias de la fig.5.6: Se ha resumido todo el algoritmo en un solo diagrama para mostrar claramente los puntos de comprobacion necesarios, aunque en una realizacion practica se habria basado la descripcion en dos tareas subordinadas:"Calculo de una coordenada en funcion de la fase" e "Incrementar modulo 256". En la parte inferior del diagrama se han numerado las ramas alternativas del algoritmo, indicando entre parentesis las que no implican ninguna accion. Tambien se han indicado los puntos de comprobacion, cuyas aserciones son:

(1) $X=Xo+Sen\ (2\pi r/256)\quad Xo-1<=X<=Xo+1$

(2) $(\exists n \mid R=mod(n*tx,\ 256)) \supset 0<=R<256$

(3) $Y=Yo+Sen(2\pi f/256) \supset Yo-1<=Y<=Yo+1$

(4) $(\exists n \mid f=mod(n*ty+fo,\ 256)) \supset 0<=f<256$

(5) $(R=0 \cap (2)) \cap (f=fo \cap (4)) \equiv$
$\equiv (\exists n, Kx, Ky \mid n*tx=Kx*256,\ n*ty=Ky*256) \equiv$
$\equiv (\exists Kx, Ky \mid Kx/tx=Ky/ty) \supset R=0, f=fo$

El simbolo $\supset$ indica que el cumplimiento de la expresion de su izquierda implica el de la de su derecha. En nuestro caso significa que la asercion de la derecha se ha obtenido simplificando la de la izquierda, ampliando asi el conjunto de valores que pueden tomar las variables, con lo que se simplifica la verificacion, aunque a costa de perder un rigor, innecesario en la mayoria de los casos.

condiciones iniciales y el agrupamiento de los caminos de cada ramificacion en las distintas verificaciones, unida a la consideracion de la condicion de final del bucle, permiten agrupar las cuatro ejecuciones previstas inicialmente en una sola con cuatro repeticiones del bucle. La tarea del ejemplo dibuja una curva de Lissajous, calculando el seno del angulo de fase a partir de una tabla que contiene los senos de 64 valores del primer cuadrante. Los caminos alternativos son consecuencia del distinto tratamiento que se debe dar a los valores de la tabla y la fase de la señal para obtener la coordenada del punto a dibujar, segun el cuadrante en que se encuantre dicha fase. En la fig. 5.6 se han indicado los caminos alternativos y las aserciones correspondientes a los puntos a trazar.

En la tabla 5.2 se muestran las agrupaciones de caminos escogidas para cada una de las 4 verificaciones necesarias. Estas agrupaciones se han hecho teniendo en cuenta que los indices de los periodos de las dos señales componentes de la figura de Lissajous seran relativamente pequeños, razon por la cual los caminos 6 y 12 se ejecutaran normalmente despues de haberse seguido los caminos 4 y 10, respectivamente, correspondientes al cuarto cuadrante. Las condiciones necesarias para la ejecucion de cada rama nos dan las condiciones que deben cumplir los valores iniciales que asignemos a R, F, Tx y Ty para que se siga la ejecucion por los caminos previstos. Como se puede apreciar, dado que al principio de la tarea se hace R=0, al iniciarse el bucle tan solo

VERIF.	CAMINOS POSIBLES												CONDICIONES INICIALES GENERALES	CONDICIONES AL FINAL DEL BUCLE
	1	2	3	4	5	6	7	8	9	10	11	12		
T1	X				X		X				X		0<=R<64 R+Tx<256 0<=f<64 f+Ty<256	R<256 f<256
T2		X			X			X			X		63<R<128 R+Tx<256 63<f<128 f+Ty<256	R<256 f<256
T3			X		X				X		X		127<R<192 R+Tx<156 127<f<192 f+Ty<256	R<256 f<256
T4				X	X					X	X		191<R<256 R+Tx<256 191<f<256 f+Ty<256	R<256 f<256

VERIF.	COND. INICIALES PARTICULARIZADAS	COND.AL FINAL DEL BUCLE
T1	R=0 Tx= 64 f=fo=0 Ty=64	R=Tx=64, f=Ty=64
T2	R=64 Tx=64 f=64 Ty=64	R=2Tx=128, f=2Ty=128
T3	R=128 Tx=64 f=128 Ty=64	R=3Tx=192, f=3Ty=192
T4	R=192 Tx=64 f=192 Ty=64	R=4Tx-256=0, f=4Ty-256=0

Tab. 5.2.- Tabla de verificacion correspondiente a la tarea de la fig. 5.6. En la tabla (a) se muestran las condiciones generales, que son las mismas impuestas por el algoritmo para que se sigan los caminos marcados con un aspa. En la Tabla (b) se han particularizado estas condiciones para conseguir que las verificaciones se encadenen automaticamente al repetirse el bucle de dibujo de un punto.

se pueden cumplir las condiciones de T1, pues el algoritmo prevee que en sucesivas repeticiones del bucle de dibujo de un punto se vayan alcanzando las condiciones de T2, T3 y T4.

La condicion inicial ideal seria aquella que repitiera el bucle tan solo 4 veces, siguiendo sucesivamente los caminos correspondientes a T1, T2, T3 y T4. Para conseguirlo debe cumplirse que al final de la primera ejecucion del bucle sea cierta la condicion inicial de T2, 64 <=R < 128, y como en el cuerpo del bucle se hace R:=R+Tx, resulta que la condicion al principio de la verificacion T1 debe ser 64 <= R+Tx < 128, la cual unida a la ya conocida 0 <= R< 64 se traduce en Tx = 64. Analogamente se demuestra que se debe cumplir Ty=64, y que estas dos nuevas condiciones son suficientes para que el bucle se ejecute de la forma deseada. Con estas consideraciones se han particularizado las condiciones iniciales expresadas en la tabla 5.2.a, para obtener las indicadas en la tabla 5.2.b., donde podemos ver que las condiciones al final del bucle en cada ejecucion son precisamente las condiciones iniciales de la ejecucion siguiente, por lo que la repeticion impuesta por el algoritmo es suficiente para ejecutar las cuatro verificaciones previstas inicialmente.

5.3.- RUTINAS DE COMPROBACION.-

El objetivo de las rutinas de comprobacion es doble. Por un lado nos deben mostrar el correcto funcionamiento de la rutina que estamos verificando. Por otro deben encargarse de imponer las condiciones iniciales escogidas para que la ejecucion sea significativa y nos permita verificar las aserciones de la tarea. Asi pues, vamos a distinguir dos tipos de rutinas de comprobacion, aunque en algunos casos una unica rutina cubra los dos objetivos.

5.3.1.- RUTINAS DE VISUALIZACION DE VARIABLES.-

Estas rutinas deben mostrar al operador los datos necesarios para que este pueda verificar el correcto cumplimiento de las aserciones de la tarea en los puntos de comprobacion. Dependiendo de la herramienta que estemos utilizando para poner a punto la tarea, esta rutina debera ser ejecutada por el mismo procesador que esta ejecutando la tarea o por otro sistema.

En el caso mas sencillo y tradicional, en el que la puesta a punto se realiza mediante un monitor o depurador residente en el sistema, las rutinas de verificacion consistiran en impresiones de las variables significativas sobre la consola, mediante llamadas a las rutinas de utilidad del sistema, a las que se transferira el contenido de registros internos posiciones de memoria, o puertas de entrada o tampones de salida, segun convenga.

Este metodo tiene el inconveniente de alterar la ocupacion de memoria y la velocidad de ejecucion de la tarea. En el caso de tareas cableadas, solo se puede emplear si la evolucion del estado de la tarea es suficientemente lenta como para que la rutina pueda ir visualizando los datos, y en todo caso implica dotar al interfaz de la tarea de la capacidad de acceder al contenido de los registros de datos y biestables de estado de la tarea.

Cuando la herramienta de puesta a punto empleada es un emulador de circuito, las rutinas de visualizacion de variables son controladas directamente por el emulador. En este caso el sistema suele contar con dos procesadores, uno encargado de emular la ejecucion de la tarea, y otro encargado de visualizar los datos pertinentes sobre la consola. Ello permite hacer ejecuciones en tiempo real, aunque entonces la visualizacion de los datos no se produce inmediatamente despues de capturarlos,sino al final de la ejecucion de la tarea, para evitar accesos multiples a la memoria de la traza. Si la tarea esta programada, el emulador permite visualizar el contenido de los registros o posiciones de memoria, despues de la ejecucion de determinadas instrucciones, pero ello implica una ralentizacion de la ejecucion de la tarea. En una ejecucion en tiempo real de tareas programadas, asi como en la verificacion de tareas cableadas, el emulador se comporta simplemente como un analizador logico.

Cuando la puesta a punto se hace mediante un

analizador logico, tan solo podremos visualizar el contenido de registros aislados, o los datos transferidos por los buses cuando se cumplen ciertas condiciones. Esto significa que no nos sera posible visualizar en cualquier momento el contenido de los registros internos del procesador, o el de posiciones de memoria. Por esta razon, en caso de que en los puntos de comprobacion determinados sea necesario verificar el contenido de registros internos, deberemos introducir en la tarea, como rutinas de visualizacion, instrucciones que transfieran el contenido de esos registros al exterior del procesador sea una posicion de memoria, sea un registro de periferico, de forma que el analizador logico pueda capturar el valor en el momento en que se transfiere a traves del bus.

Las rutinas cableadas puestas a punto mediante un analizador logico, deberan incluir operadores para obtener la señal del reloj y los cualificadores de esta, para fijar los instantes en que el analizador debe capturar los datos almacenados en los registros de la tarea. En estos casos ademas existe la limitacion impuesta por el numero de señales binarias que es capaz de almacenar o trazar el analizador, las cuales limitan el numero de variables que se puedan verificar en los puntos de comprobacion correspondientes a cada ejecucion. Esto implica que cuando el conjunto de variables a verificar entre todos los puntos de comprobacion tenga mas bits que el analizador, deberemos hacer la verificacion por partes, comprobando en cada ejecucion de la tarea un subconjunto de las variables de tamaño adecuado, a las posibilidades del analizador o

emulador empleado.

5.3.2.- RUTINAS DE INICIALIZACION.-

Dado el proceso ascendente seguido en la verificacion de las tareas programadas, la rutina de inicializacion de la tarea a verificar se limita a reproducir el proceso de llamada de la rutina, con los parametros necesarios para establecer las condiciones iniciales determinadas para cada una de las ejecuciones de verificacion.

Si se sigue el orden de puesta a punto recomendado, las rutinas de inicializacion siempre estaran programadas, pues la primera tarea a verificar sera la cableada de nivel mas alto o la programada de nivel mas bajo, de forma que, en cualquier caso, el nivel inmediatamente superior corresponderá a tareas programadas, y en el deberemos incluir temporalmente a la rutina de inicializacion. A esta primera tarea se iran añadiendo el resto de las tareas a verificar, formandose un nucleo de tareas que ira creciendo hacia los niveles superiores e inferiores. La incorporacion de tareas de nivel superior, al nucleo de tareas verificadas, se hara sustituyendo la rutina de inicializacion por la nueva tarea a verificar mas su rutina de inicializacion.

La verificacion de las tareas del nivel mas bajo del nucleo requiere, ademas de la rutina de inicializacion, una serie de rutinas que simulen las tareas de niveles

inferiores llamadas por la tarea en trance de verificacion, o incluso el entorno de trabajo del sistema si es que aun no se ha conectado este a los perifericos que debe controlar. Todo ello debido a que las tareas cableadas se iran incorporando al sistema tanto mas tarde cuanto mas "lejos" se encuentren del procesador, es decir, cuanto mas bajo sea su nivel y mas proximas se encuentren a los perifericos.

5.4.- TRAZA DE LA TAREA.-

Por traza de la tarea entendemos el rastro que las sentencias de comprobacion ejecutadas nos dejan sobre la herramienta de puesta a punto que estemos empleando. Esta definicion concreta algo mas el concepto de traza introducido en el capitulo 3 al hablar de los analizadores logicos y emuladores de circuito, donde se decia que la traza era simplemente el rastro dejado por la tarea en la memoria del analizador, y que por tanto solo coincidira con la definicion dada ahora si se utiliza alguna caracteristica exclusiva de las sentencias o rutinas de comprobacion para cualificar el reloj de muestreo.

Vamos a analizar los tres tipos de traza que podemos obtener de un analizador o emulador, segun se seleccione o no la informacion disponible, y segun se interfiera o no en la normal ejecucion de la tarea.

5.4.1.- TRAZA PASO A PASO.-

Esta primera forma de obtener la traza se caracteriza por proporcionar una informacion exhaustiva, pues nos indica el estado de todos los registros internos del procesador tras la ejecucion de cada instruccion. Tan solo se debe utilizar en la verificacion de tareas programadas en ensamblador, cuando se sospecha que algun codigo de instruccion de un tramo concreto del programa no corresponde con el listado o es ejecutado erroneamente por

el procesador. En general se debe recurrir a la ejecucion paso a paso cuando se plantea algun problema inprevisto a nivel de ejecucion de una instruccion, como pueden ser los problemas relativos a los punteros del procesador, los señalizadores de la unidad aritmetico-logica, etc.

En las tareas cableadas no tiene mucho sentido el concepto de ejecucion paso apaso, pues supondria el bloqueo del reloj tras cada ciclo, para verificar el contenido de todos los registros, lo cual eliminaria la mayoria de los problemas dinamicos, producidos precisamente por los retardos introducidos en las señales, cuando estos se hacen del orden de magnitud del periodo del reloj. Tambien se podria considerar como traza paso a paso de una tarea cableada, la obtenida tomando muestras cada ciclo del reloj de mayor frecuencia de la tarea, sin emplear cualificadores, con lo que se obtendria una traza de todas las posibles variaciones de los registros, supuesto el sistema sincromo y salvo impulsos aleatorios, que no todos los sistemas de verificacion permiten detectar.

5.4.2.- TRAZA EN PARALELO.-

Es la mas usada y se obtiene como resultado de aplicar exactamente el metodo de verificacion propuesto; esto es programando la herramienta de verificacion para que nos muestre la traza de las variables significativas, es decir las que afectan a las aserciones

correspondientes a los puntos de comprobacion. Es aplicable a cualquiera de los metodos de realizacion de tareas. Este tipo de traza se caracteriza por ejecutar las rutinas de comprobacion en paralelo con las acciones propias de la tarea, sin necesidad de detener la ejecucion de esta, como sucedia en la ejecucion paso a paso. Segun sea el tipo de herramienta de verificacion empleado y el soporte las variables a verificar, la obtencion de la traza de paralelo supondra o no perdida de tiempo durante la ejecucion de la tarea.

Cuando se emplea un depurador, o cuando las variables significativas estan almacenadas en registros internos del procesador, debemos añadir a la tarea unas rutinas de visualizacion que supondran una perdida de tiempo durante la ejecucion.

Si se emplea un analizador logico o un emulador, podemos evitar la insercion de las rutinas de visualizacion, obteniendo los valores de las variables significativas directamente de los buses del computador cuando se modifica su valor.

5.4.3.- TRAZA EN TIEMPO REAL.-

Como su propio nombre indica, consiste en obtener la traza de la tarea sin alterar en absoluto la velocidad de ejecucion de esta. Realmente, en los casos en que es posible obtener la traza en tiempo real esta puede

coincidir con la traza en paralelo, tan solo diferiran en los casos en que se hayan incluido rutinas de visualizacion, las cuales en este caso no estan permitidas, pues alteran la velocidad de ejecucion. Cuando las variables significativas son modificadas sin que su contenido se haga accesible desde los buses externos a la pastilla del procesador, no podremos obtener su traza en tiempo real, y deberemos conformarnos con observar la evolucion del algoritmo, constatando simplemente la secuencia de ramas de este que se ejecutan y los resultados finales.

La obtencion de la traza en tiempo real tan solo es necesaria en la verificacion de tareas que se encuentren en el nivel inmediatamente superior al de la barrera circuito/programa, es decir aquellas que utilizan directamente tareas cableadas. De entre estas cabe destacar especialmente aquellas en las que el sincronismo con la tarea cableada es critico, sea por la cantidad de señales a manipular, sea por la complejidad de las restricciones inpuestas por el protocolo de la comunicacion entre ambas tareas, sea por la necesidad de controlar o generar temporizaciones precisas.

5.5.- CASOS PARTICULARES.-

Hasta ahora hemos supuesto que las tareas a trazar podian estar realizadas por programa o por circuito, pero en todo caso controladas directamente por el procesador principal, y sincronizadas con el a traves de su reloj. En estos casos el uso de un analizador logico o un emulador para la puesta a punto de la tarea es indiferente, salvo las ventajas ya mencionadas, del primero frente al segundo. Existen otros medios de realizacion de tareas en los que estas condiciones no se cumplen, por lo cual el emulador de circuito no es utilizable, ya que la evolucion de la tarea deja de estar sincronizada con el procesador principal; sea porque evoluciona mas rapidamente que el reloj de este, sea porque esta controlada por otro procesador.

5.5.1.- TAREAS MAS RAPIDAS QUE EL PROCESADOR.-

Este es uno de los casos en los que las ventajas del emulador se convierten en inconvenientes, pues al capturar los datos del bus del procesador sincronamente con este, no puede obtener informacion de lo que sucede entre dos flancos activos del reloj. En estos casos el analizador logico se convierte en la unica herramienta util para la puesta a punto, pues podemos sincronizarlo con el reloj propio del circuito, si este es sincrono, o combinar los distintos impulsos de reloj, si el circuito fuera asincrono. Asi pues, salvo la descalificacion del

emulador, la puesta a punto de estas tareas es identica a la de las normales.

5.5.2.- TAREAS REALIZADAS POR PROCESADORES ESPECIALIZADOS.-

Tal vez el caso mas peculiar, de todos los que pueden presentarse en la realizacion de una tarea del sistema, sea aquel en el cual la tarea es realizada por otro procesador. En realidad es tan solo un caso particular de la posibilidad, ya mencionada, de microprogramar una tarea realizada por circuito. La peculiaridad del caso consiste en el hecho de emplear una maquina de aplicacion general para controlar la ejecucion del algoritmo, por lo cual el microprograma se convierte en un programa para el nuevo procesador y la tarea cableada, desde el punto de vista del sistema global, se convierte en programada desde la optica del nuevo subsistema.

No vamos a realizar un analisis exhaustivo del problema planteado por esta especial forma de realizar una tarea, pues requeriria una extension desproporcionada en el contexto de esta obra; tan solo vamos a apuntar las soluciones a aplicar en algunos de los casos que se pueden presentar.

Cuando la complejidad de una tarea a cablear recomienda la introduccion en el sistema de un procesador adicional, la capacidad de control del sistema queda dividida. Ello obliga a replantear la distribucion de las

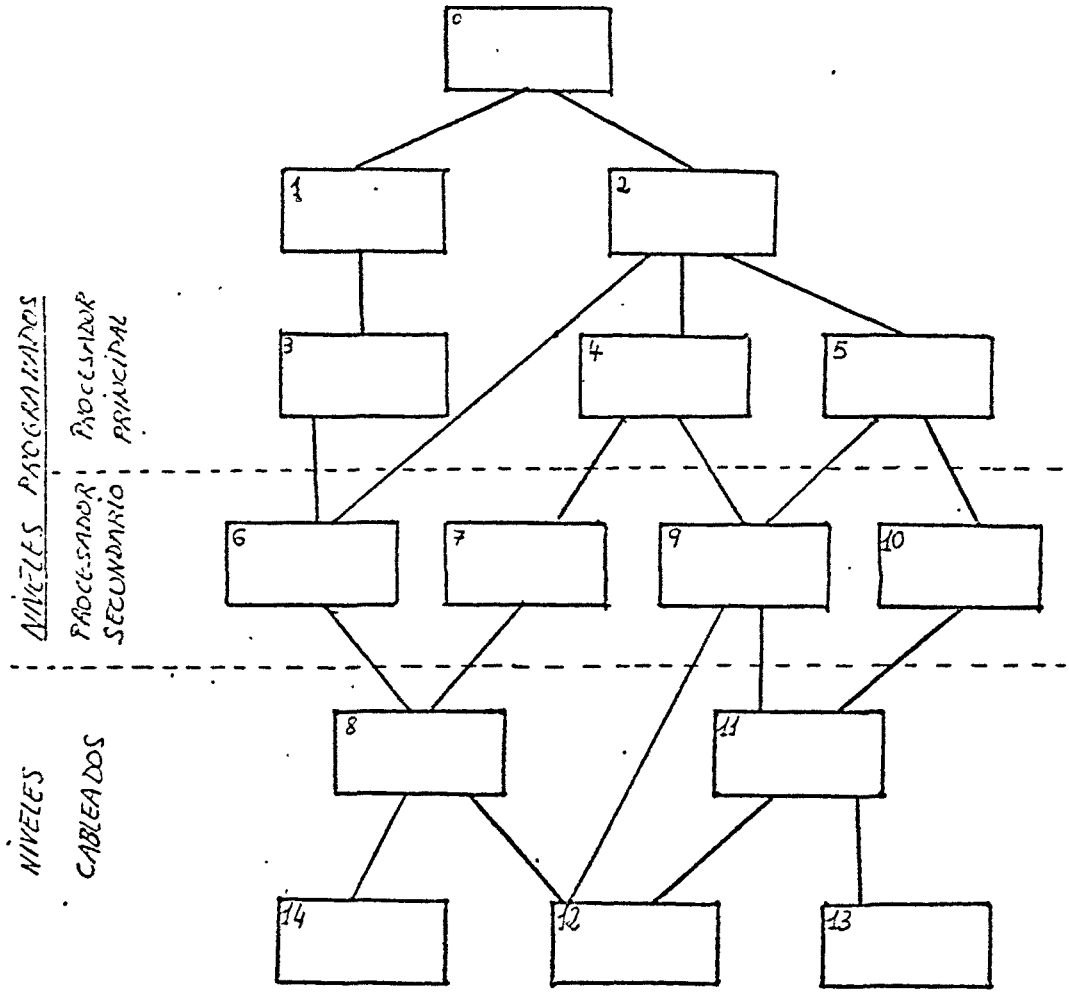


Fig. 5.7.- Organigrama jerarquico de un sistema biprocesador, en el cual el procesador secundario controla todas las tareas cableadas.

tareas en el organigrama jerarquico, desdoblando la barrera programa/circuito en dos, entre las cuales se encontraran las tareas programadas en el procesador secundario, pues normalmente una sola tarea no saturara su capacidad, y aprovecharemos su potencia asignandole otras tareas inicialmente cableadas o programadas.

Si el procesador secundario acapara el control de todas las tareas cableadas (fig.5.7), la jerarquizacion de los procedimientos de realizacion sigue siendo total. Tan solo habra problemas de reentrancia a nivel de la comunicacion entre ambos procesadores, si admitimos la ejecucion en paralelo de ambos procesadores, lo cual puede provocar llamadas a tareas del procesador secundario antes de que haya finalizado la ejecucion concurrente de la llamada anteior. Ello unido a la necesidad de unificar los mecanismos de comunicacion entre los procesadores hara necesario introducir una tarea de gestion de la comunicacion y planificacion de las tareas del procesador secundario, la cual normalmente sera ejecutada por este.(fig.5.8).

Si el sistema admite una descomposicion en modulos relativamente independientes, de forma que podamos asignar a uno de los procesadores las tareas correspondientes a un modulo (fig.5.9), el problema se simplifica; pues las interacciones y comunicaciones entre procesadores seran minimas.

Finalmente nos encontramos con el caso mas general y

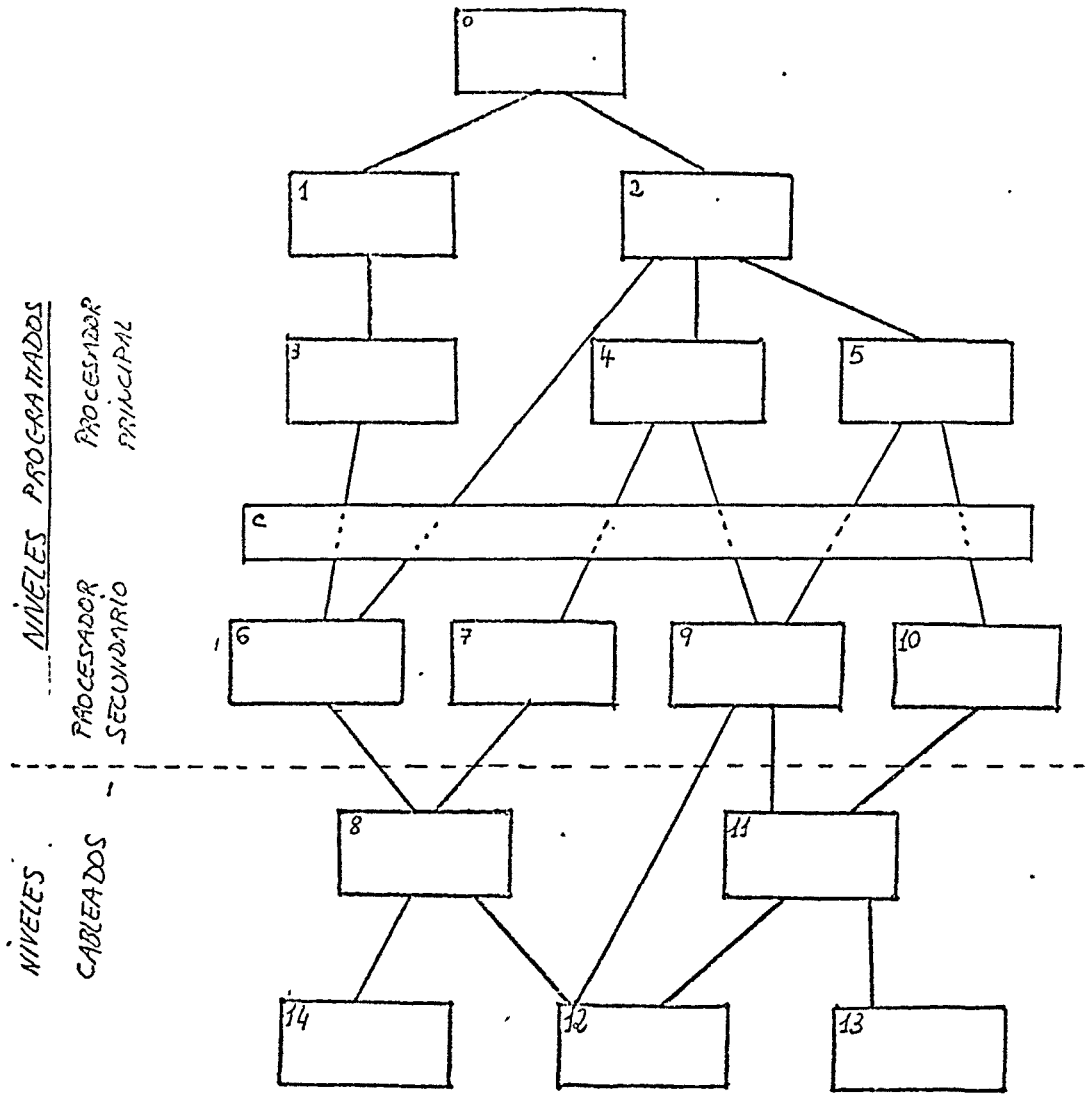


Fig. 5.6.- Organigrama jerarquico resultante de introducir una tarea de planificacion del trabajo (c) en el organigrama de la fig. 5.7. Ello permite independizar completamente las tareas del procesador principal que se comunican con el secundario, con lo cual ambos pueden estar multiprogramados sin problemas de sincronismo.

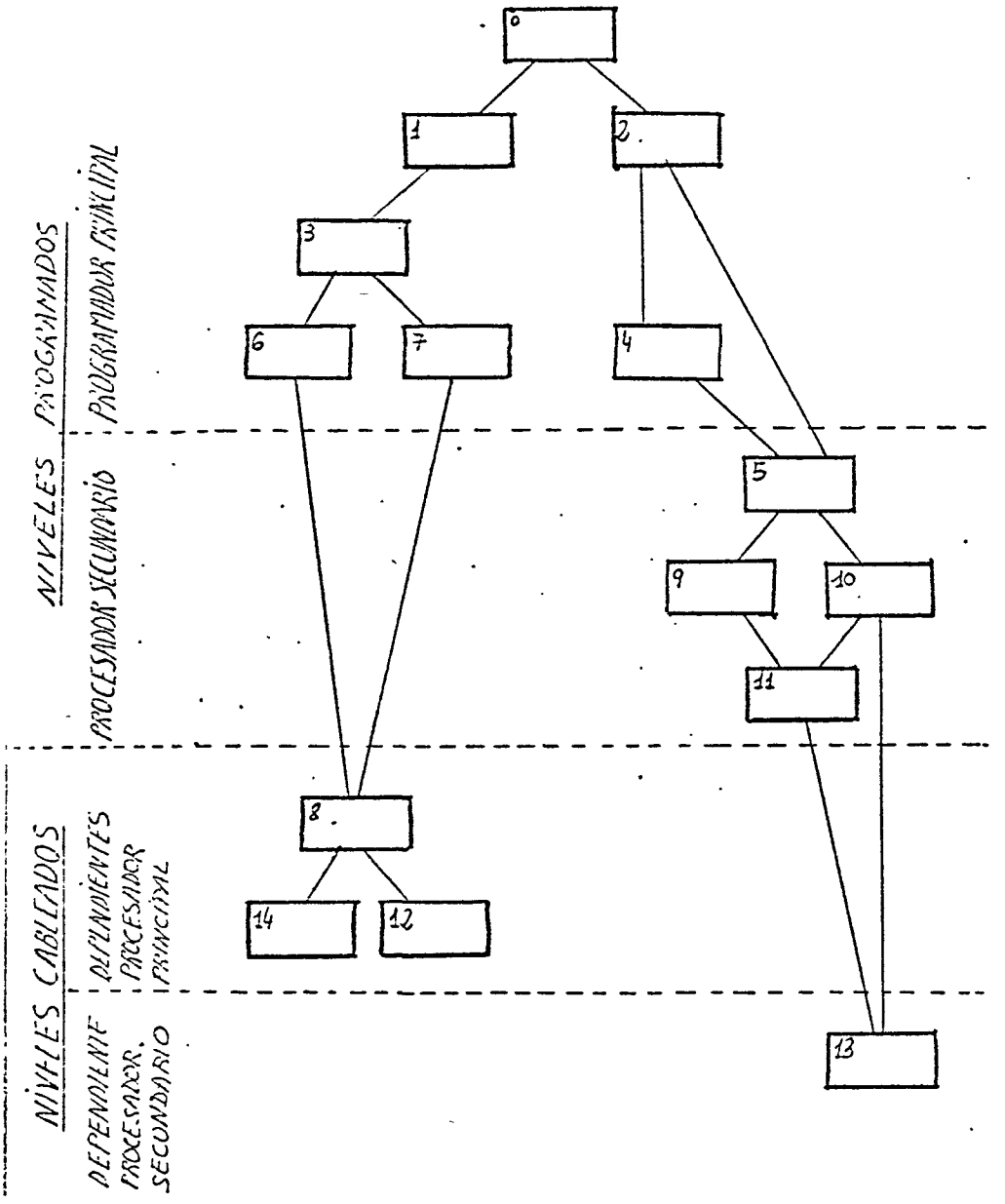


Fig. 5.9.- Organigrama jerarquico de un sistema biprocesador, que admite una descomposicion en modulos relativamente independientes. En este caso ambos procesadores tienen acceso a las tareas cableadas, pero como no comparten ninguna, no hay problemas de sincronismo.

mas complejo, desde el punto de vista organizativo, consistente en asignar al procesador secundario tan solo parte de las tareas que utilizan alguna tarea de los niveles inferiores, normalmente cableados (fig.5.10). Entonces se plantea un problema de concurrencia al compartir ambos procesadores algun recurso comun. Este hecho complica, tanto las tareas que usan una tarea compartida, como las propias tareas compartidas, obligando a emplear señalizadores que indiquen el estado de ocupacion de las tareas accesibles a ambos procesadores, con posibilidad de verificacion y activacion (Test and set) en un solo ciclo de acceso. Si las tareas compartidas son mas de una, se pueden producir "abrazos mortales" que es necesario prevenir y evitar.

Desde la optica de la puesta a punto, este ultimo caso plantea el problema de necesitar dos rutinas de inicializacion para las tareas compartidas, que deben ejecutarse en cada uno de los procesadores simultanea y sincronamente, de forma que podamos verificar las protecciones de la tarea frente a accesos multiples simultaneos o casi-simultaneos. En este caso puede resultar util la introduccion en las tareas de inicializacion de un temporizador comun a ambas y legible simultanea e independientemente por ambos procesadores, lo que permitira sincronizar ambas tareas de inicializacion para verificar todos los casos posibles de acceso a la tarea compartida.

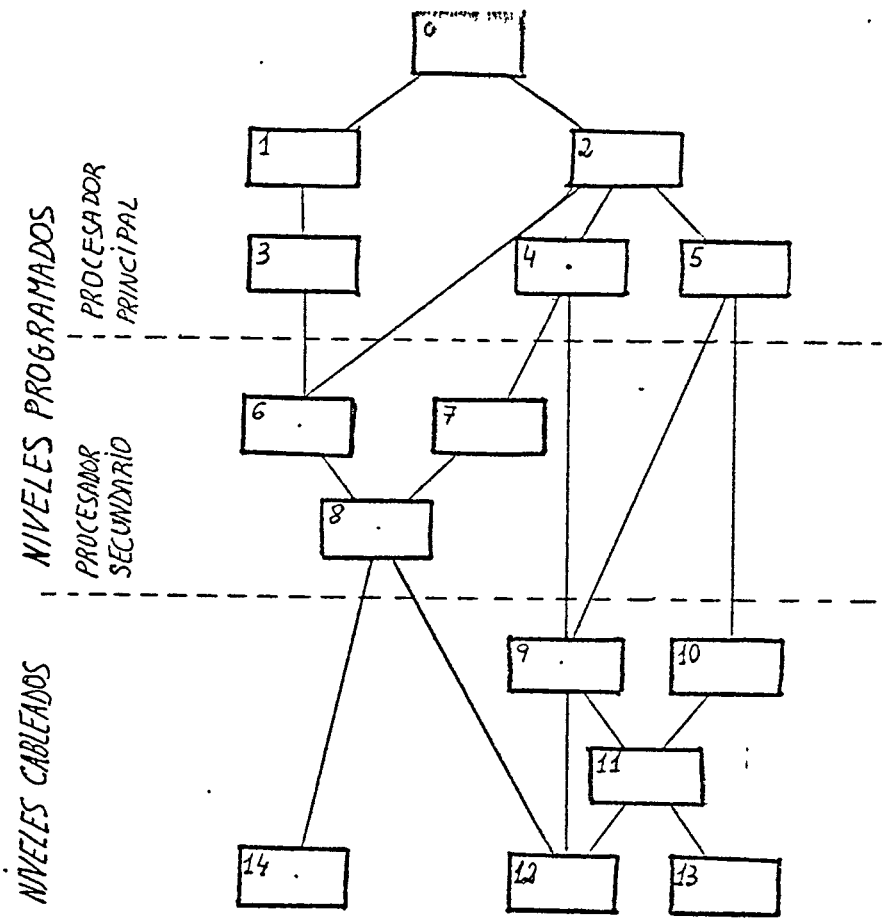


Fig. 5.10.- Organigrama jerarquico de un sistema biprocesador, en el cual ambos procesadores comparten el control de las tareas cableadas. Este es el caso mas general, pues hay una tarea a la cual tienen acceso ambos procesadores, por lo que se deberan prever los posibles accesos simultaneos.

RESUMEN DEL CAPITULO

Para la puesta a punto de sistemas basados en microprocesador se ha adoptado el criterio de aplicar un procedimiento no exhaustivo y orientado fundamentalmente a la puesta a punto de sistemas de envergadura relativamente pequeña. Esta restriccion ha permitido descartar los sistemas de verificacion automatica de programas o de simulacion de circuitos, pues requieren el concurso de maquinas de gran potencia, que no estaran al alcance de la mayoria de los diseñadores. Por el contrario se ha desarrollado un metodo de puesta a punto especialmente orientado al empleo de analizadores logicos y emuladores de circuito, herramientas estas que se estan convirtiendo en algo tan usual como un osciloscopio.

El fundamento del metodo de puesta a punto, reside en la determinacion de un minimo de puntos de comprobacion del algoritmo. De esta forma la verificacion del correcto funcionamiento de la tarea se reduce a comprobar el estado del sistema al ejecutarse las instrucciones correspondientes a esos puntos de comprobacion. Para simplificar esta verificacion se determinan las condiciones iniciales necesarias para ejecutar al menos una vez cada uno de los puntos de comprobacion, pero el minimo numero de veces necesario. El estado del sistema en estos puntos se caracteriza mediante sus aserciones, las cuales indican los conjuntos de valores posibles de las variables significativas en ese momento de la ejecucion del algoritmo, lo cual es muy util para programar un

trazado selectivo de la ejecucion de la tarea mediante un analizador logico, evitando el almacenamiento en su memoria de informaciones irrelevantes.

CONCLUSIONES DE LA PARTE II

El analisis de los metodos de trabajo o empleados hasta el momento nos ha permitido constatar uno de los supuestos de partida, que justificaban este trabajo: La falta de conexion y compatibilidad entre las metodologias de diseño de programas y circuitos. La primera consecuencia de esta disparidad de procedimientos era la falta de conexion entre las descripciones y documentaciones de las partes programada y cableada, y con la consiguiente dificultad a la hora de comprender las interrelaciones entre ambas, cada dia mayores. La concepcion global del sistema y su descripcion descendente, empleando un unico lenguaje para todas las partes en las cuales se iba descomponiendo, se ha demostrado que era factible y perfectamente util para cubrir este primer objetivo del trabajo. La descripciones de los algoritmos ofrecen una vision global de todas las funciones realizadas por el sistema, y de las interacciones entre estas, a traves de los argumentos transferidos de una a otra.

La aplicacion de esta tecnica general a las tareas cableadas y programadas, unidas a la sistematizacion de los mecanismos de comunicacion entre ambos tipos de tareas, hacen que las interacciones entre las partes cableada y programada del sistema queden automaticamente definidas, cubriendose asi el segundo objetivo propuesto.

Respecto al problema de la sintesis, se ha demostrado

la posibilidad de trazar un unico camino de diseño, siguiendo una secuencia de objetivos gradualmente alcanzables, aplicable a todas las tareas del sistema, con independencia del procedimiento de realizacion final. Este camino nos va suministrando descripciones de la tarea literalmente traducibles a lenguaje de alto nivel estructurado, lenguaje de alto nivel no estructurado lenguaje de bajo nivel, o lenguaje descriptor de circuitos o diagrama de bloques logicos. Se han dado ejemplos que demuestran efectivamente la posibilidad de realizar automaticamente estas traducciones con lo cual podemos afirmar que el cambio en el procedimiento de realizacion de una tarea significa simplemente avanzar o retroceder en la sucesion de descripciones. De esta forma queda perfectamente cubierto el tercero de los objetivos del trabajo presentado, pues no solo se han unificado los metodos de concepcion y diseño sino que se han hecho coincidir en uno solo.

Dado que el diseño de un sistema no concluye hasta que se ha verificado su correcto funcionamiento, no se podia olvidar este aspecto en el proceso de verificacion de metodologias. En el ultimo capitulo de la obra se ha propuesto un procedimiento de puesta a punto de circuitos y programas, basado en el analisis de la descripcion general de cada tarea. Puesto que todas las tareas han sido descritas siguiendo un mismo criterio, la planificacion de la puesta a punto se llevara a cabo con independencia del procedimiento de realizacion elegido para la tarea. La puesta en practica de esta planificacion

dependera evidentemente del procedimiento de realizacion y de las herramientas disponibles, pero en todo caso se ha demostrado la utilidad del metodo propuesto al aplicarlo mediante las herramientas unas modernas para la puesta a punto: Analizadores Logicos y Emuladores de Circuito.

Finalmente, para demostrar la consecucion del cuarto objetivo de esta obra, se han planteado una serie de ejercicios a los alumnos de las asignaturas de la catedra, y se ha podido observar como la aplicacion de la metodologia propuesta les permitia realizar unos diseños bien documentados y verificar sencillamente el correcto funcionamiento de los sistemas, con independencia del procedimiento de realizacion y de las herramientas del diseño empleadas.

P A R T E III A P E N D I C E S

A P E N D I C E I

LENGUAJE DESCRIPTOR DE ALGORITMOS: PASCAL-NATURAL.-

Tal como se indica en el texto, este lenguaje no significa mas que un ejemplo de la forma que debe tener un lenguaje de descripcion de algoritmos. En esta version se han reducido al minimo las palabras llave de las sentencias, para simplificar la escritura y la legibilidad de la descripcion. Todo ello a costa de asociar a cada nivel de sentencias estructuradas un nivel de tabulacion o indentacion distinto, con lo que se evitan las ambiguedades que pudieran presentarse en la interpretacion de la descripcion.

El lenguaje se ha definido bajo la forma de Backus- Naur (BNF) adaptada a las imposiciones de la impresion mecanografica sencilla, es decir suprimiendo los superindices y subindices por dos formas de parentesis distintas, segun los casos:

[] indica aparicion opcional del bloque, ninguna o a lo sumo una vez.

{} implica repeticion opcional del bloque, entre 0 y n veces.

La descripcion es la siguiente:

<Tarea> ::= <cabecera tarea>
<cuerpo tarea>
FIN

<cabecera tarea> = TAREA <identificador tarea>[(<lista argumentos>)]

<cuerpo tarea> = {<sentencia>}

<sentencia> = <sentencia sin etiqueta>
|<etiqueta><sentencia sin etiqueta>

<sentencia sin etiqueta> = <sentencia simple>
|<sentencia estructurada>

<sentencia simple> = <sentencia de asignacion>
|<sentencia de salto>
|<llamada a tarea>
|<sentencia vacia>

<sentencia asignacion> = <variable> = <expresion>

<llamada a tarea> = <identificador tarea> [(<lista param.act.>)]

<lista param.act.> = <parametro actual>{ <parametro actual>}

<sentencia de salto> = SALTA <etiqueta>

<sentencia estructurada> = <sentencia condicional>
| <sentencia repetitiva>
| <sentencia con>

<sentencia condicional> = <sentencia si>|<sentencia caso>

<sentencia si> = SI <expresion>
<tab> ENTONCES <sentencia>
{<tab> <tab> <sentencia>}

[<tab> SINO <sentencia>

{<tab> <tab> <sentencia>}]

<tab> .= TABULADO|'   '

<sentencia caso> = CASO <expresion> DE

{<tab> <elemento lista de casos>}

<elemento lista casos> = <lista etiquetas casos>:<sentencia>

{<tab> <sentencia>}

<sentencia repetitiva> = <sent.mientras>|<sent.repetir>|<sent.para>

<sentencia mientras> = MIENTRAS <expresion> HACER

{<tab> <sentencia>}

<sentencia repetir> = REPETIR <sentencia>

{<tab> <sentencia>}

HASTA <expresion>

<sentencia para> = PARA <variable control>.=<lista para>HACER

{<tab> <sentencia>}

<lista para> = <valor inicial> SUBE <valor final>[SALTO <valor increm.>]

| <valor inicial> BAJA <valor final>[SALTO <valor increm.>]

<sentencia con> = CON <lista de variables registro> HACER

{<tab> <sentencia>}

<parametro actual> = <variable>|<expresion>

<variable> = <identificador>|<variable de registro>
|<variable de matriz>

<identificador tarea> = <identificador>

<etiqueta> = <identificador>

<lista etiquetas casos> = <identificador caso>{,<identificador caso>}
| OTRO

<identificador caso> = <constante>

<variable control> = <variable>

<valor inicial> = <variable>|<expresion>

<valor final> = <variable>|<expresion>

<valor incremento> = <variable>|<expresion>

<lista variables registro> = <variable de registro>
{ <variable de registro>}

<variable de registro> = <identificador registro>.<campo>

<identificador registro> = <identificador>

<campo> = <identificador>

<variable de matriz> = <identificador> (<lista indices>)

<lista indices> = <indice> {,<indice>}

<indice> ::= <expresion>

<expresion> .= <variable>|<constante>
|[<operador>]<expresion>{<operador><expresion>}

<variable> := <constante>|<identificador>

<constante> = <identificador>|<numero>|<cadena>

<identificador> = {<caracter>}

<cadena>::= {<caracter>|<signo puntuacion>}

<caracter>::= A|B|.. |Z|a|b|. |z|_

<signo puntuacion> = !|.. |@

<numero> .= [<signo>]{<cifra>}

<signo> = + | -

<cifra>::= 0 | . . | 9

A P E N D I C E II

LENGUAJE DE TRANSFERENCIA DE REGISTROS: LTR

Permite describir las transferencias entre los registros internos o posiciones de memoria del procesador que controla el sistema, o entre los registros particulares de una tarea cableada. Supone implicitamente que las sentencias se ejecutaran secuencialmente en el orden en que han sido escritas, salvo indicacion contraria de una sentencia de salto.

La descripcion en LTR de una tarea comprende tres bloques:

- Definicion de elementos de memorizacion y terminales.

- Descripcion de operadores combinacionales.

- Secuencia de sentencias.

A continuacion se detallan los elementos que pueden emplearse en cada uno de estos tres bloques de la descripcion.

1.-DEFINICIONES:

Registro.. = R[N:0] Registro R de n+1 bits

Subregistro::= RI[n:0]=CI[n:n1] CO[n1-1:0] :Reg.RI compuesto por dos subregistros CI y CO de n-n1+1 y n1 bits respectivamente.

Memoria::= M[p:0,n:0] memoria con p+1 posiciones de n+1 bits cada una.

Biestable::= C registro de 1 solo bit.

Entrada = E[n 0]

Salida::= S[n:0]

2 -OPERANDOS:

Registro completo: = R, RI, CI, CO.

Parte de un registro::= R[n1.n0]

Varios registros encadenados: = CI CO

Partes de registros encadenadas: = R[n1 0] RI[n:n0]

Posicion m de una memoria: = M[m]

Bit n de un registro::= R[n]

Bit n de la posicion m de una memoria = M[m:n]

3 - OPERADORES:

Aritmeticos: +, -, *, /, .MOD. 7.MOD.3=1

Logicas: = .AND.= Interseccion= .Y.

= OR. = Union= .O.

= XOR.= Union exclusiva= .OEX.

= NOT.= Negacion o complementacion= .NO.=X

Desplazamiento:

.desi.= despl.a la izq.=> R[i]<-R[i-1], R[0] <-0 (i=n.1)

desd.= despl.a la der.=> R[i]<-R[i+1], R[n] <-0 (i=n-1...0)

.desad.= despl.arit.der.=> R[i]<-R[i+1], R[n] <-R [n]

(conserva el bit de signo)

ciri.= circ.a la izq.=> R[i] <- R[i-1], R[0] <-R[n]

.cird.= circ.a la der => R[i] <- R[i+1], R[n] <-R[0]

4 -OPERACIONES:

4 1.- TRANSFERENCIA IMPERATIVA:

A<-B => El registro A se llena con el contenido del reg.B

A<-A+B => El registro A se llena con resultado de sumar los contenidos de los registros A y B.

A<-.ciri.A => A se llena con el resultado de circular a la izq.A.

4.2.- TRANSFERENCIA CONDICIONAL:

SI C ENTONCES CP <- CD

[SINO CP <- CP+1]

(Si el biestable C vale 1 ENTONCES se llena CP con el contenido de CO.
si C=0 ENTONCES se llena CP con el resultado de sumarle 1.)

5.- SENTENCIAS.-

Estan formadas por una operacion precedida o no de una etiqueta, segun su ejecucion pueda producirse o no despues de una sentencia distinta de la que la precede inmediatamente. Normalmente la operacion indicada sera una transferencia unica o una lista de transferencias separadas por comas, si se pueden llevar a cabo todas ellas simultaneamente, o en un orden indefinido. Pero la operacion tambien puede ser de ruptura de secuencia:

IRA <etiqueta>

Esta operacion referenciada en forma condicional permite transcribir a LTR cualquier algoritmo:

SI <condicion> ENTONCES IRA <etiqueta si>
SINO IRA <etiqueta no>

A P E N D I C E III

LENGUAJE ENSAMBLADOR UNIFICADO: CALM

Este lenguaje recopila las instrucciones de los microprocesadores mas comunes. Frente a los ensambladores propuestos por los fabricantes tiene la ventaja de emplear los mismos nemotecnicos para las instrucciones similares, sea cual sea el procesador que la ejecute. Ha sido propuesto por J D Nicoud, en las revistas Euromicro Newsletter y Microscope.

Los convenios adoptados son los siguientes:

1.- REGISTROS -

Contador de Programa	PC
Puntero a la pila (16 bits)	SP
Registro indice (16 bits)	XX, IX. IY
Registro de 8 bits	A, B. R0, R1, R2. .
Registro de 16 bits	HL, IX . P0, P2. .
Registro de indic. (flag.)	F

Indicadores	Acarreo C, enlace L, signo S, Desbordamiento V, cero Z, Interrupcion I, modo decimal D.

2 - REFERENCIAS A MEMORIA.-

Posicion de direccion m	m
Pos. cuya direccion es el contenido de x	(x)
Pos. cuya direccion esta en la pos. de direccion m	@m

3 - EXPRESIONES NUMERICAS.-

4 bits	i
Entero sin signo de 8 bits	n
Entero con signo de 8 bits	n'
Entero sin signo de 16 bits	m
Entero con signo de 16 bits	m'

4 - OPERADORES -

Suma	add	+

Resta	sub	-
Multiplicacion	mul	*
Division	div	/
Negacion	not	-
Y	and	^
O	or	v
O-Exclusiva	xor	

5 - PREFIJOS DE INSTRUCCIONES.-

Operandos invertidos	I
Operacion decimal	D
Operacion de coma flotante	F

6 - SUFIJOS DE INSTRUCCIONES.-

Con acarreo	C
Con bit de enlace	L

Operacion con octeto

(en microprocesador de 16 bits) B

Seguida de salto (jump) si t es cierto JT

Seguida de esquive (Skip) si t es cierto ST

7 - MODOS DE DIRECCIONAMIENTO -

7 1 - INMEDIATO:

Toma la forma: # <valor>

Donde <valor> puede ser cualquier expresion numerica sin signo.

7 2.- IMPLICITO.-

Emplea un simbolo predefinido, correspondiente a un registro:

RO, A, XX...

7 3.-ABSOLUTO -

Hace referencia a una direccion de memoria. Puede tomar las formas:

Direccion completa (14 a 16 bits) m

Direccion de la pagina 0 (8 bits) n

Direccion de la misma pag.que la instr (8 bits) !n

7 4.- RELATIVO.-

Tambien hace referencia a una direccion de memoria, pero especificando un desplazamiento a partir del valor del contador de programa:

Desplazamiento de 8 bits con signo	(PC)+n' o .+n'
Desplaz. positivo de 8 bits en la misma pag.	(PC)!+n o .!+n
Desplaz. de 16 bits con signo	(PC)+m' o .+m'

7 5.- INDEXADO.-

Referencia una posicion dse memoria, tomando como base un registro indice:

Sin desplazamiento	(XX)
Desplazamiento de 8 bits positivo	(XX)+n
Desplazamiento de 8 bits con signo	(XX)+n'
Desplazamiento de 8 bits con signo	(XX)+m'
Desplazamiento de 8 bits con signo en pag.0	(X)+n'
Desplazamiento de 8 bits en la misma pagina	(X)!+n
Desplazamiento de 16 bits con signo	(X)+m'

El registro indice puede ser de 16 bits (XX), o de 8 bits (X). Si su contenido se incrementa o decrementa automaticamente se indicara:

Post incrementado	(XX+)
Post decrementado	(XX-)
Pre incrementado	(+XX)
Pre decrementado	(-XX)

7 6.- INDIRECTO.-

Se referencia la posicion de memoria en la que se encuentra la direccion del operando. La indireccion se caracteriza mediante el simbolo @ precediendo la direccion:

Indirecto absoluto	@m
Relativo indirecto	@(PC) + n' o @.+ n'
	@(PC) + m' o @.+ m'
Indexado indirecto (pre-indexado)	@(XX) + n'
	@(XX) + m
	@(XX) + m'
Indirecto indexado (post-indexado)	(X) + @n
	(XX) + @m

8 - CONDICIONES DE EJECUCION:

Las condiciones generalmente admitidas en las instrucciones de ruptura de secuencia condicional son:

Acarreo a uno	C=1	CS	o	LO
Acarreo a cero	C=0	CC	o	HS
Bit de signo a uno	S=1	SS	o	MI
Bit de signo a cero	S=0	SC	o	PL
Bit de cero a uno	Z=1	ZS	o	EQ
Bit de cero a cero	Z=0	ZC	o	NE
Desbordamiento a uno	V=1	VS		
Desbordamiento a cero	V=0	VC		

Despues de comparar o restar dos numeros positivos:

Mayor (>)	C.O.Z=o	HI
Mayor o igual (>=)	C=o	HS
igual (=)	Z=o	EQ
menor o igual (<=)	C.O.Z=1	LS
menor (<)	C=1	LO

Despues de comparar o restar dos numeros codificados en complementoa dos (con signo):

Mayor (>)	Z.O.(S.OEX.V)=0	GT
Mayor o igual (>=)	-(S.OEX.V)=0	GE
Igual (=)	Z=0	EQ
Menor o igual (<=)	Z O.(S.OEX.V)=1	LE
Menor (<)	S.OEX.V=1	LT

9 - NEMOTECNICOS.-

Los nemotecnicos de las instrucciones ejecutadas por la mayoria de los microprocesadores son:

Cargar d con s	d <- s	LOAD	d s
Intercambiar d y s	d<-> s	EX	d s
Salvar d en la pila	(-SP)<-d	PUSH	d
Cargar d con el tope de la pila	d<-(SP+)	POP	d
Sumar d con s, resultado en d	d<-d+s	ADD	d s
Sumar con acarreo d con s	d<-d+s+C	ADDC	d.s
Sumar en decimal	d<-d+s	DADD	d s
Sumar en decimal con acarreo	d<-d+s+C	DADDC	d.s
Restar	d<- d-s	SUB	d.s
Restar con acarreo	d<-d-s-C	SUBC	d s
Resta invertida	d<- s-d	ISUB	d.s
Sumar el complemento a uno	d<- d+-s	ACO	d.s
Sumar el completo a 1 con acarreo			
Suma invertida del complemento a 1	d<- s+-d	IACO	d.s

Multiplicar	d<- d*s
Dividir	d<- d/s
Y logico	d<- d.Y s
Poner a cero bits	d<- d.Y -s
O logico (Poner a uno bits)	d<- d.O.s
O exclusivo	d<- d.OEX.s
No exclusivo	d<- d OEX.-s
Comparar (restar sin cargar)	d-s
Comprobar bit (Y sin carga)	d Y s

Rotar a la derecha d

Rotar a la izquierda d			
Rotar a la derecha sobre el acarreo			
Rotar a la izquierda sobre el acarreo			
Desplazar a la derecha			
Desplazar a la izquierda			
Despl.a la der. sobre el acarreo			
Despl.a la izq. sobre el acarreo			
Despl. aritmetico a la derecha			
Despl. aritmetico a la der. sobre el acarreo			
Intercambiar los cuartetos			
Alto (espera interrupcion externa)			
Restituir dispositivos externos.			
Poner a 0 d	d<- 0	CLR	d
Poner a 0 bit i de d	d(i)<- 0	CLR	d:i
Poner a 0 indicador b	b<- 0	CLRb	
Poner a 1 bit i de d	d(i)<- 1	SET	d:i
Poner a 1 indicador b	b<- 1	SETb	
Pulsar b	b<- 0	PULb	
Complementar a 1 d	d<- -d	CPL	d
Negar d (compl.a 2)	d<- -d	NEG	d
Valor absoluto de d	d<-/d/	ABS	d
Ajuste decimal tras suma		DAA	d
Ajuste decimal tras resta		DAS	d

Ajuste decimal (suma o resta)		DA d
Increnetar d	d<-d+1	INC d
Incr. d y saltar si t es cierta		INCJ,t d,a
Incr. d y esquivar si t es cierta		INCS t d,a
Decrementar d	d<- d-1	DEL
Decr. d y saltar si t es cierta		DECJ,t d a
Decr. d y esquivar si t es cierta		DECS,t d,a
Comprobar t (cargar indicadores)	d<- d	TEST d
Comp. bit i de d (carga indicador Z)		TEST d:i
Saltar (incondicionalmente) a a		JUMP a
Saltar si t es cierta		JUMP,t a
Esquivar la siguiente instruccion		SKIP
Esquivar la siguiente instr.si t es cierta		SKIP,t
Esquivar dos palabras de memo.		LSKIP
LLamar a subrutina en a		CALL a
LLamar a subrutina en a si t es cierta		CALL,t a
Volver de subrutina.		RET
Volver si t es cierta		RET t
Trap (al vector de interrupcion v)		TRAP (v)
Volver de interrupcion		RTI
Ejecutar una sola instruccion en a		EXEC A

Habilitar interrupciones (poner a 0 mascara)	ION
Inhabilitar interr. (poner a 1 mascara)	IOF
No operacion	
Retardo proporcional al valor de	DELAY d
Espera una interrupcion	WAIT

B I B L I O G R A F I A

1.- A.ALABAU, J.FIGUERAS: "An Algorithm for Boolean function implementation using a Microprocessor". Proc.Euromicro Workshop. Nice june 1975; pag. 259-261.

2.- A. ALABAU, J.FIGUERAS: "Some aspects of the implementation of sequential systems on a Microcomputer" Proc. Symposium MIMI'75.Zurich june 1975; pag. 162-166.

3.- A.ALABAU, J.FIGUERAS: "A Hard-Softw. Microcomputer course in electrical engineering curriculum". Computer Science and Engin curricula workshop, IEEE Comp.Soc. Virginia, June 1977.

4.- J.FIGUERAS, A.ALABAU: "An experience on C education" Euromicro Newsletter. vol.3.nb.2, apr.'77; pag.34-38.

5.- M.MEDINA, M.VALERO: "Top-down methodology to design I/O systems".Journees d'Electronique de L'E.P.F.Laussanne. '79.

6.- DIJKSTRA: "A discipline of programming". Prentice-Hall, Ser.in Automatic Computation.1976.

7.- JACKSON: "Principles of program design". Academic Press. London 1975.

8.- WIRTH: "Algorithms + Data Structures = Programs" Prentice Hall 1976.

9.- NICOUD: "Traite d'electricite". Vol.14. Ed. Georgi.

10.-STOCKENBERG, DAM: "Vertical Migration for Performance Enhancement in Layered Hardw/Firmw/Softw. Systems". Computer IEEE may 1978.

11.-NASSI-SHNEIDERMANN: "Flowchart Techniques for Strutured Programming". ACM Sigplan vol. 8-8.Aug. 1973.

12.-MANO: "Computer System Architecture". Prentice-Hall. 1976.

13.-CHU: "Computer Organization and Microprogramming". Prentice Hall. 1972.

14.-HILL, CHU, DIETMEYER, SIEWIOREK: "Introducing Computer Hardware description Languages". Computer IEEE. Dec.1974.

15.-HARTENSTEIN: "Fundamentals of Structured Hardware Design". North-Holland. 1977.

16.-K.JENSEN, N.WIRTH: "PASCAL user manual and report" Springer-Verlag 1978.

17.-LEE.SAMUEL C.: "Microcomputer design and Applications"

Academic Press 1977.

18.-WASSERMAN.A.I.: "A Top.Down View of Software Engineering". (IEEE). Procedings of the 1 Nat.Conf. on Software Eng.'75.

19.-ALLAN, S.J., OLDEMOEFT, A.E.: "Aflow Analysis Procedurefor the translation of high-level languages to a Data flow lang." IEEE. Trans.Computers.Sept.'80.

20.-ARSENAULT J.E., ROBERTS J.A.: "Reliability and Maintainability of Electronic Systems". Comp. Science Press.1980 (USA).

21.-RONBACK, J.A.: "Software Reliability-How it affects System Reliability". Proced 1975 SRE Canadian Reliability Symposium. Pergamon Press.NY.pp 125-127.

22.-MILLER, J.R.: "Program Testing". Computer, Apr'78. pp. 10-12.

23.-TURSKI,W.M.: "Computer Programming Methodology" MEYDEN.'78.

24.-ALAGIC, S., ARBIB, M.A.: "The Design of well-Structured and correct programs".

25.-DIJKSTRA, E.W.: "A Constructive approach to the problem of program correctness" Bit. vol.8, n. 174. (1968).

26.-ELSPAS.B. et al.: "An assessmont of techniques for proving program correcthess". A@M Comput. Surv. vol.4, 98 (1972).

27.-YODER, Cornelia M., SCHRAG, Marilynl.: "Nassi-Schneidermann charts an alternative to flowcharts for design." Proc. of Software Quality and Assurance Workshop. (1978).

28.-WIRTH, Niklaus: "Systematic Programming: An Introduction". Prentice-Hall. 1973.

29.-"PLZ Operators manual". ZILOG.Co.

30.-DIJKSTRA, HOARE, DAHL: "Structured Programming". Academic Press. London 1972.

31.-BASS, C.: "Microsystems; PLZ-A family of system programming microprocessors". COMPUTER IEEE.Mar. 1978.

32.-BROMW W.: "Microsystems: Modular programming in PL/M". COMPUTER IEEE. Mar.1978.

33.- "MPL reference manual". MOTOROLA.

34.- "FD1771 Application note". WESTERN DIGITAL

35.-WORDEN, J.: "Design considerations for dual-density diskette controllers". COMPUTER DESIGN. Jun. 1978

36.-WAKERLY, J.F.: "Logic design projects". J.Wiley & Sons. 1976.

37.-GOOS, G.: "Software engineering: Hierarchies". 1975 Springer-Verlag. Lecture Notes in Computer Science n.30.

38.-McGOWMAN & KELLY: "Top-Down Structure programming techniques". Petrocelli/Charter. New York 1975.

39.-NICHOLS, J.L.: "Hardware versus software for microprocessors I/O". COMPUTER DESIGN.Aug.1976.

40.-WAKERLY, J.F.: "Documentation Standars clarify design". COMPUTER DESIGN. Feb.1977.

41.-PROSSER F.& WINKEL, D.: "Mixed Logic leads to maximun clarity with minimum hardware". COMPUTER DESIGN. May 1977.

42.-Jesus GALVAN y Manel MEDINA, "A new method for the electromagnetic optimization of stereotaxically implanted electrodes in the brain", en Med. Progr. Technol., n.3, pg. 181 a 186, (1976).

43.-Manel MEDINA y J.Daniel NICOUD, "Simple cassette interface", en MICROSCOPE, N.7, vol.1, (1977).pag.5.31-1 a 5.31-10.

44.-Manel MEDINA y Antonio ALABAU, "Interconexion de un

conjunto de microcomputadores a un miniordenador para compartir recursos". Comunicacion presentada a las Jornadas Univem '78, organizadas por la Fundacion Universidad-Empresa. (17pp.)

45.-Catedra de Ordenadores de la ETSIT.UPB., Diversas colaboraciones en la seccion fija "Micromundo" de la revista MUNDO ELECTRONICO, años 1978 a 1980.

46.-Manel MEDINA, "Lenguajes de alto nivel para microprocesadores I y II", en MUNDO ELECTRONICO, n.84 y 85. (1979) (10 pp.)

47.-Manel MEDINA y Otros, "Microprocesadores y Microcomputadores". Cap. 14, de la 3a. edicion (8pp.). Marcombo 1980.

48.-Manel MEDINA y otros, "Interconexion de perifericos a microprocesadores". Caps. 3 y 10 (42 pp). Marcombo 1980.

49.-Manel MEDINA y Mateo VALERO, "Top/Down Methodology to design I/O Systems".Comunicacion presentada a las "Journees d'Electronique 1979", organizadas por la E.P.F.L. de Lausanne (Suiza).

50.-Manel MEDINA Y Mateo VALERO, "Diseño de una interfase para cassettes analogicos". Apuntes 3er. curso de Microcomputadores. ETSIT. UPM. 1979.