ERSITAT 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

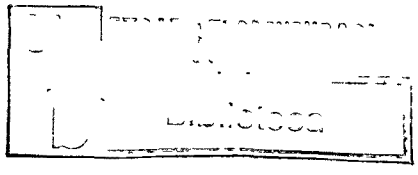
DIRIGIDA POR :

ANTONIO ALABAU MUÑOZ

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

Esta tesis doctoral fué leída el día 15 de junio de 1981 en la Escuela Técnica Superior de Ingenieros de Telecomunicación, de la Universidad Politécnica de Barcelona, ante el tribunal compuesto por:

Elías Muñoz Merino (presidente)
Jesús Galván Ruiz
Joan Figueras Pamies
Angel Cardama Aznar
Antonio Alabau Muñoz (secretario)

Dicho tribunal calificó este trabajo con la nota de Sobresaliente "Cum Laude".

A Manuel y Agustina

por su confianza y desvelos.

A Assumpcio

por su comprension.

AGRADECIMIENTOS

A Antonio Alabau, por haberme orientado en los dificiles momentos de la eleccion del camino a seguir, ayudado a conseguir el irprescindible scporte tecnico a lo largo de ese camino y animado a afrontar con decision la recta final.

A Joan Figueras, por haberme ayudado en la determinacion del punto de partida mis trabajos.

A Jesus Galvan, por haberme encaminado hacia la programacion de ordenadores.

A Elias Muñoz Merino, por haber aceptado la presidencia del tribunal encargado de juzgar este trabajo.

A Angel Cardama por haber accedido a juzgar el interes de esta obra.

A Francisco Cancillo, por haberme brindado la oportunidad de confrontar los metodos propuestos en esta obra, cuando tan solo eran timidas ideas, colaborando de esta forma a perfilar los aspectos mas importantes de las mismas.

A Mateo Valero, por haberme ofrecido un banco de prueba adecuado para la presentacion de este metodo a nivel internacional.

A Isabel Gallego, por haberse prestado a emplear una de las primeras versiones de este metodo en sus primeros diseños, ayudandome de esa forma a detectar muchas de las lagunas cubiertas por la version aqui presentada.

A Josep Posch y Celesti Rosello por haber mentenido el soporte fisico necesario para confeccionar el original de este ejemplar, y al resto de miembros del departamento por su colaboracion en la presentacion del trabajo.

A los alumnos de ordenadores de 1980/81, por haberme ayudado a demostrar la utilidad practica y didactica del metodo, al aplicarlo en la descripcion y puesta a punto de sus diseños.

Y finalmente a Maite Medina, por haber convertido los garabatos surgidos de mi pluma en simbolos inteligibles.

I N D I C E

2.- HERRAMIENTAS PARA EL DISEÑO DE PROGRAMAS.-

2.1.- PROGRAMACION ESTRUCTURADA.-

2.1.1.- Caracteristicas generales.

2.2.- HIPO.-

2.2.1.- Fases del desarrollo de un programa.

2.2.2.- Herramientas de analisis y sintesis.

2.2.3.- Procedimiento mental de diseño.

2.2.4.- Beneficios obtenidos.

2.3.- METODOLOGIA JACKSON.-

2.3.1.- Procedimiento de diseño.

2.3.1.1.- Fases.

2.3.1.2.- Objetivos.

2.3.1.3.- Aspectos contemplados.

2.3.2.- Estructuras de flujo.

2.3.2.1.- Secuencia.

2.3.2.2.- Iteracion.

2.3.2.3.- Seleccion.

2.4.- DESCRIPCION DE ALGORITMOS.-

2.4.1.- Program Design Languaje: PDL.

2.4.2.- PASCAL.

2.4.3.- Diagramas de Nassi-Schneidermann.

3.- HERRAMIENTAS PARA LA PUESTA A PUNTO DE SISTEMAS.-

3.1.- OBJETIVOS GENERALES DE LA PUESTA A PUNTO.-

3.1.1.- Procedimientos para la verificacion de programas.

3.1.1.1.- Exhaustiva.

3.1.1.2.- Selectiva.

3.1.1.3.- Formal.

3.1.1.4.- Automatica.

3.1.2.- Herramientas automaticas para la verificacion.

3.1.2.1.- Analisis estatico del programa fuente.

3.1.2.2.- Analisis dinamico del programa fuente.

3.1.2.3.- Mantenimiento del programa.

3.1.2.4.- Mejora de las prestaciones.

3.2.- PROGRAMAS DEPURADORES (Debuggers).-

3.2.1.- Visualizacion de la memoria.

3.2.2.- Visualizacion de los registros.

3.2.3.- Puntos de ruptura de programa (Breakpoints).

3.2.4.- Desensamblado y ensamblado de instrucciones en linea.

P A R T E II

ANALISIS DE LA METODOLOGIA PROPUESTA.-

4.- METODO BARNA PARA EL DISEÑO DE SISTEMAS BASADOS EN MICROCOMPUTADOR.-

4.1.- ANALISIS DESCENDENTE.-

4.1.1.- Descomposicion del problema.

4.1.1.1.- Funcional.

4.1.1.2.- Modular.

4.1.2.- Concepto de analisis descendente.

4.2.- JERARQUIZACION DE LAS TAREAS.-

4.2.1.- Organigrama Jerarquico.

4.2.2.- Niveles Jerarquicos.

4.2.3.- Criterios para la agrupacion en niveles.

4.2.3.1.- Complejidad de las estructuras de datos.

4.2.3.2.- Proximidad a los perifericos.

4.2.3.3.- Complejidad de las estructuras de control.

4.3.7.- Traduccion a lenguaje de bajo nivel (LBN).

4.3.7.1.- Ubicacion de las variables.

4.3.7.2.- Adaptacion de los modos de direccionamiento.

4.3.7.3.- Adaptacion de las estructuras de control.

4.3.7.4.- Codificacion de las expresiones: Operadores.

4.3.7.4.1.- Codificaion en linea.

4.3.7.4.2.- Macroinstruccion.

4.3.7.4.3.- Subrutina.

4.3.7.5.- Sincronizacion con tareas de nivel inferior.

4.3.8.- Traduccion a circuitos.

4.3.8.1.- Sincronizacion con las tareas de nivel superior.

4.3.8.2.- Discriminacion de las partes de control y de tratamiento de datos.

4.3.8.3.- Descripcion en diagrama de bloques de circuitos.

4.3.8.3.1.- Bloques de tratamiento.

4.3.8.3.2.- Bloques de control.

4.3.8.3.3.- Bloques funcionales.

4.3.8.4.- Descripcion en LDM.

4.4.- RECONVERSION DE PROGRAMA A CIRCUITO.-

4.4.1.- Utilidad de la reconversion.

4.4.2.- Caso del circuito a medida.

4.4.3.- Caso del circuito general particularizado.

4.4.4.- Paralelismo en la ejecucion de las tareas.

4.5.- RECONVERSION DE CIRCUITO A PROGRAMA.-

4.5.1.- Utilidad de la reconversion.

4.5.2.- Aprovechamiento de la potencia sobrante de un procesador secundario.

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

5.1.- CONSIDERACIONES GENERALES.-

5.1.1.- Procedimientos de validacion.

5.1.1.1.- Orden de la validacion.

5.1.1.2.- Profundidad de la validacion

5.1.2.- Aprovechamiento de la documentacion.

5.1.3.- Seleccion de los puntos de comprobacion.

5.1.4.- Eleccion de los utiles de trabajo.

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

5.2.1.- Determinacion de los caminos a ejecutar.

5.2.2.- Determinacion de las aserciones.

5.2.3.- Determinacion de los puntos y variables a trazar.

5.2.4.- Seleccion del conjunto de condiciones iniciales.

5.3.- RUTINAS DE COMPROBACION.-

5.3.1.- Rutinas de visualizacion de variables.

5.3.2.- Rutinas de inicializacion.

5.4.- TRAZA DE LA TAREA.-

5.4.1.- Traza paso a paso.

5.4.2.- Traza en paralelo.

5.4.3.- Traza en tiempo real.

5.5.- CASOS PARTICULARES.-

5.5.1.- Tareas mas rapidas que el procesador.

5.5.2.- Tareas realizadas por procesadores especializados.

INDICE DE FIGURAS Y TABLAS.-

Fig. 5.9.- Organigrama jerarquico de un sistema biprocesador modular.

Fig. 5.10.- Organigrama jerarquico de un sistema biprocesador modular.

INTRODUCCION GENERAL.-

Desde la aparicion de los microprocesadores en España en 1974, se han ido introduciendo cada vez mas en dos mundos hasta entonces practicamente inconexos, desde el punto de vista del usuario: El proceso de datos y los sistemas logicos. De esta forma se ha ido perfilando un puente entre ambos, el cual ya no permite establecer alegremente la tradicional distincion entre diseñadores de programas y diseñadores de circuitos, pues los sistemas basados en microprocesadores incluyen ambos aspectos, y al menos parte de los responsables de un proyecto deberan conocer las herramientas y metodos de diseño de ambas partes del sistema.

Al laboratorio de ordenadores de la E.T.S.E.T. le han preocupado desde un principio, tanto el conocimiento de las posibilidades de los microprocesadores aplicados al diseño de sistemas digitales de control, como la formacion de profesionales capaces de afrontar con exito la nueva problematica de diseño. Por esta razon se comenzo por una primera fase de formacion del profesorado: A nivel teorico, consultando la bibliografia aportada por los congresos pioneros en el tema, como EUROMICRO, y a nivel practico, realizando cursillos y estancias de formacion en Lausanne (Suiza). Esta fase permitio poner a punto los primeros sistemas basados en microprocesador, que sirvieron para iniciar las primeras experiencias de enseñanza de los mismos a los alumnos, y permitio tambien realizar proyectos basados en aplicaciones reales, en

estrecha colaboracion con empresas del sector electronico.

La experiencia adquirida en estos primeros años de trabajo con microprocesadores, unida a la realimentacion obtenida de los alumnos que han seguido los cursos para postgraduados impartidos por la catedra: Ecole d'Ete du Forez (Francia) y Escola d'Informatica d'Estiu (Barcelona); nos ha permitido detectar la necesidad de ofrecer una alternativa unificada para los metodos dispares de diseño empleados por los programadores y los diseñadores de circuitos. De esta forma se conseguiria simplificar el "curriculum" de los diseñadores de sistemas basados en microprocesadores.

Nuestro trabajo ha consistido en buscar una solucion que cumpla los requisitos impuestos por la mencionada alternativa, y en determinar el conjunto minimo de conocimientos que se debe impartir a un futuro diseñador de sistemas basados en microprocesador.

Una vez perfilado el metodo de diseño alternativo, lo hemos puesto en practica, tanto en la realizacion de proyectos del laboratorio (ver bibliografia), como en la programacion de los cursos de ordenadores de la Escuela, apreciandose un considerable aumento del rendimiento de los alumnos que lo han empleado en la realizacion de las practicas de laboratorio.

El objetivo de este trabajo es mostrar la metodologia de diseño de sistemas basados en microprocesador, la cual

abarca los siguientes conceptos:

* Descripcion, global del sistema.

* Descripcion de las interacciones entre circuitos y programas que componen el sistema.

* Unificacion de los metodos empleados para la concepcion y descripcion de circuitos y programas en los sucesivos pasos de refinamiento.

* Especificacion de cada una de las fases del diseño, de forma que el proceso sea comprensible y reproducible por los diseñadores noveles o inexpertos.

De esta forma el trabajo tiene dos aspectos. De una parte tenemos una contribucion tendente a mejorar los metodos de diseño empleados hasta ahora. De otra parte tenemos el aspecto didactico del metodo, pues ofrece una alternativa a los cursos de diseño convencionales, en los que los aspectos circuito y programa son tratados con total independencia, con la consiguiente dificultad a la hora de encontrar una metodologia para la concepcion de los sistemas completos.

Nuestro metodo se fundamenta en tres principios:

* Concepcion descendente del sistema, basada en refinamientos sucesivos de las descripciones.

* Estructuracion del diseño de cada una de las tareas en que se va descomponiendo al sistema.

* Descripcion en lenguaje de transferencia de registros (RTL) de las tareas realizables por circuito o programadas en ensamblador, indistintamente.

Esta memoria de Tesis comprende tres partes bien definidas:

1) Descripcion de los metodos y herramientas de diseño empleadas hasta ahora en la concepcion y puesta a punto de programas y circuitos. En los tres capitulos que comprende esta parte se analizan y resumen los metodos mas empleados para:

- Diseño de circuitos digitales.
- Diseño estructurado de programas.
- Puesta a punto de circuitos y programas.

Puntualizando, finalmente las ventajas e inconvenientes planteados por cada uno de ellos, extrayendo asi las caracteristicas esenciales del metodo de diseño propuesto.

2) Descripcion del metodo de diseño descendente, llamado Barna en honor a la ciudad en que ha sido ideado. Como ya se ha dicho, el metodo se fundamenta en dar el mismo trato a las partes programadas y cableadas del sistema. Por esta razon se ha optado por una expresion cronologica de las fases del diseño:

- La primera comprende el analisis del problema, la descomposicion en tareas y la descripcion de los algoritmos para su resolucion. En esta fase no se realiza absolutamente ninguna distincion en funcion del procedimiento de realizacion escogido para una tarea.

- La segunda abarca la sintesis o realizacion de las tareas, que se lleva a cabo por refinamientos sucesivos de las descripciones hasta llegar a un nivel adecuado al procedimiento de diseño escogido para cada tarea. De esta forma los distintos procedimientos se corresponden a distintos peldaños de una misma escala que une las descripciones mas general y mas detallada, del algoritmo.

- La tercera afronta el problema de verificacion y puesta apunto de las tareas del sistema, y es donde se pueden observar las principales ventajas del sistema, al permitirnos sistematizar con gran facilidad la estrategia de verificacion de una tarea, con independencia de la forma de realizacion. Esta tan solo influira en la herramienta de verificacion empleada.

3) Finalmente se incluyen unos ejemplos de los lenguajes de descripcion de las tareas empleados en los distintos niveles de detalle del diseño del sistema. Estos apendices describen:

- Un lenguaje de descripcion de algoritmos a nivel general.

- Un lenguaje de transferencia de registros, para la descripcion al nivel mas bajo previo a la realizacion en ensamblador o circuito.

- Un lenguaje ensamblador comun a los microprocesadores mas corrientes, que evita confusiones en la interpretacion de los nemotecnicos.

A R T E I

ETODOS DE DESCRIPCION Y
ERRAMIENTAS DE DISEÑO
L SISTEMAS

NTRODUCCION A LA PARTE I

Como ya hemos dicho, en esta parte se incluye una ecopilacion de los metodos de diseño empleados hasta el omento por los programadores y por los diseñadores de istemas digitales, asi como una descripcion de las erramientas de puesta a punto de programas y circuitos. odo ello nos permitira presentar el "estado del arte" en os dos campos englobados por los sistemas basados en icroprocesador, y nos permitira detectar las ventajas e nconvenientes planteados por cada herramienta al plicarla en concreto a los microprocesadores.

Mientras los microprocesadores se aplicaban casi xclusivamente al diseño de sistemas digitales de control, ustituyendo la circuiteria discreta de control de ircuito, la complejidad de los programas era elativamente pequeña, ello permitia a los diseñadores frontar el problema sin una metodologia de programacion specifica. Este hecho ha contribuido a que apenas se ayan producido progresos en este campo, pues se han apli- cado las tecnicas ya empleadas en la programacion de

ordenadores grandes y minis, aprovechandose directamente las experiencias de programacion estructurada conocidas.

Respecto al diseño de los circuitos perifericos del microprocesador, tampoco ha sido necesaria la aplicacion de una metodologia especifica, pues en realidad el microprocesador simplificaba la parte de control, al integrarla toda en una sola pastilla. Asi pues, hasta el momento ha sido suficiente el empleo de diagramas de bloques y lenguajes descriptores de circuitos. Tan solo la puesta a punto de los nuevos sistemas ha exigido una remodelacion de las herramientas empleadas, pues el aumento en el nivel de integracion de los circuitos restringe considerablemente la accesibilidad al estado interno del sistema, lo cual obliga a emplear herramientas que permitan almacenar la informacion en el momento en que esta accesible, por si interesa consultarla mas adelante. Tal es el caso de los analizadores logicos y los emuladores de circuitos.

El empleo de todas estas herramientas no es imprescindible en el diseño de sistemas sencillos, pero la aparicion de los microprocesadores de 16 bits, la sofisticacion de los juegos de instrucciones de los de 8 bits, y la implantacion cada vez mayor de sistemas multimicroprocesadores, hacen que el diseñador eche de menos una metodologia que le permita afrontar los nuevos problemas, inabordables sin un microprocesador, con el maximo de garantias de exito. De ahi la necesidad de conocer todas estas herramientas y la forma de aplicarlas

coherentemente al diseño de sistemas basados en microprocesador, aprovechando las ventajas aportadas por cada una de ellas. Ventajas estas que iremos destacando en cada capitulo de esta primera parte.

En esta primera parte incluimos tres capitulos:

En el primero resumimos algunos de los lenguajes descriptores de circuitos que nos han servido de base para la descripcion de los sistemas basados en microprocesador a nivel de transferencia de registros.

En el segundo resumimos las metodologias de analisis y definicion de programas mas empleadas, y que hemos incluido en el metodo de diseño propuesto; y describimos las principales caracteristicas de los lenguajes unas comunmente empleadas en la descripcion de algoritmos, los cuales nos han servido de referencia para la definicion del lenguaje propuesto por nosotros para la descripcion del sistema basado en microprocesador, a nivel de algortimo.

En el tercero, finalmente, resumimos brevemente las tecnicas de verificacion de programas mas conocidas, las cuales hemos adaptado a las necesidades especificas de verificacion de sistemas basados en microprocesador; y tambien describimos las caracteristicas de las herramientas disponibles para la puesta a punto de sistemas basados en microprocesador, desde las mas tradicionales como los monitores o depuradores, hasta las mas modernas como los emuladores de circuito.

1.- LENGUAJES DESCRIPTORES DE CIRCUITOS.-

INTRODUCCION.-

Como ya se ha dicho, en este capitulo se pretenden analizar las caracteristicas de las herramientas empleadas por los diseñadores de circuitos digitales, para concebir, describir y documentar dichos circuitos. De todo el conjunto de posibilidades existente, se han seleccionado los metodos mas estructurados y susceptibles de ser simulados, puesto que uno de los objetivos de la obra es encontrar un punto de confluencia entre los metodos de diseño de circuitos y programas. Esta restriccion inicial nos encamina hacia los lenguajes de descripcion de circuitos u ordenadores. Es decir, aquellos lenguajes que han sido definidos con una cierta gramatica, y que son susceptibles de ser simulados con mas o menos facilidad.

De todos los lenguajes existentes para la descripción de sistemas logicos (DDL, CDL, IPS, HDL, RTL, CHDL....) hemos escogido dos: CDL Y KARL. Son los que consideramos mas utiles e interesantes para el diseño de sistemas basados en microprocesador. El primero (CDL), porque es el mas difundido, y es relativamente sencillo encontrar un simulador que nos permita verificar el correcto funcionamiento de los sistemas antes de montarlos.
Por ello se ha tomado como referencia para los otros lenguajes descritos.

El segundo lenguaje descrito es KARL. Este lenguaje tambien se puede simular, pero admite la realizacion de descripciones simplificadas para el caso de que no se desee simular el circuito. Estas descripciones presentan la ventaja de ser muy comprensibles y faciles de realizar.

Del analisis de estos lenguajes deduciremos su posible utilizacion en la descripcion de las operaciones realizadas tanto por el procesador (programas), como por los circuitos perifericos diseñados especificamente para la aplicacion, pues en definitiva ambas se pueden considerar como transferencias entre registros.

1.1.- LENGUAJE DESCRIPTOR DE ORDENADORES: CDL.-

Este lenguaje fue propuesto por YAOHAN CHU en 1968. Es altamente descriptivo para identificar elementos tales como: Registros, decodificadores, memorias, indicadores luminosos y terminales. Permite describir algoritmos y operaciones, a nivel de bitio, palabra o tabla de bitios, y tambien señales de temporizacion y ordenes de control. El CDL permite describir transferencias y operaciones en paralelo. Tiene el inconveniente de estar excesivamente orientado a la simulacion, lo que hace que la traducion de la descripcion CDL a circuito o viceversa, no sea inmediata, lo cual hace que su utilidad se limite, casi exclusivamente a los casos en los que se cuente con un ordenador para la simulacion y posterior diseño del circuito equivalente.

El lenguaje describe los circuitos mediante unas sentencias de declaracion, seguidas de unas microsentencias, que se organizan en secuencias.

1.1.1.- SENTENCIAS DE DECLARACION.-

Estas sentencias, no solo identifican los elementos del circuito, sino que tambien dan un nombre simbolico a cada uno de los elementos y, si es necesario, especifican sus funciones.

1.1.1.1.- SENTENCIA REGISTRO.-

Se distinguen cinco tipos de declaraciones.

* Registro (Register): Declara el nombre del registro y el rango de variacion de los subindices que corresponden a cada bit, caso de ser mas de uno.

Register, A(7 0), E

* Subregistro (Subregister): Declara un subconjunto de los bitios de un registro, previamente declarado, con una identidad propia.

Subregister, A(OP)=A(7 5), A(DIR)= A(4 0)

* Cascada de registros (Casregister): Declara la concatenacion de dos o mas registros o subregistros, previamente definidos, con una entidad propia.

Casregister, BA(8-0)=B A(7 0)

* Matriz de registros (Array register): Declara un registro bidimensional:

Array register, C(7 0,3 0)

* Memoria (Memory): Declara una matriz de registros cuyos elementos sae pueden referenciar mediante una direccion.

Memory, M(A)=M(255 0, 7 0)

1.1.1.2.- SENTENCIAS DE PERIFERICO.-

Sirven para describir los elementos del circuito que interaccionan con su entorno. Son dos tipos, segun la comunicacion sea desde o hacia el circuito:

* Interruptor (switch): Permite la entrada de datos desde el exterior del circuito. Puede tener una o varias posiciones o estados. Si tiene una posicion, sera un pulsador o un impulso; si son varias las posiciones, sera un conmutador o un conjunto de lineas biestables.

switch, POWER(ON), ACCUM(A.B)

* Indicador luminoso (light): Permite la salida de datos hacia el entorno del circuito. Sus posibilidades son las mismas que en un interruptor.

Light, FIN(SI.NO), SALTA(SI)

1.1.1.3.- SENTENCIAS TERMINALES.-

Sirven para describir redes logicas compuestas exclusivamente por elementos combinatorios. En esta descripcion se admiten los operadores basicos de la tabla 1 1., en la que ademas de los operadores basicos del algebra de Boole, se han incluido los mas usuales de los circuitos logicos. Un circuito semi-sumador seria:

Terminal, C=A*B, S=A B

1.1.1.4.- SENTENCIA DE RELOJ.-

Describen el nombre del reloj y las fases de que consta. Caso de que sean mas de una, se suponen igualmente desfasadas y en cualquier caso periodicas.

Clock, P(1 3)

1.1.2.- MICROSENTENCIAS.-

Son las sentencias destinadas a describir las microoperaciones. Una microoperacion es elemental y funcional, fisicamente construida en un circuito digital. En general una microsentencia consiste en la transferencia del resultado de evaluar una expresion, en la que intervienen unos operadores y unos operandos. Los operadores validos son los mismos que en las sentencias terminales, ya descritos.

1.1.2.1.- EXPRESIONES.-

Indican las operaciones que se deben realizar con los datos, antes de efectuarse la transferencia indicada en la microoperacion. Los operandos pueden ser constantes o variables, y formados por uno o mas bitios.

Las constantes tienen siempre subindice si estan en octal, o si estan en binario y se presta a confusion. Las constantes decimales y las binarias que no presentan duda, nunca llevan subindice.

Ejemplos de expresiones son:

x' shl Y(ADDR) X Y 3shlx 70 "y

1.1.2.2.- MICROOPERACIONES.-

Una microoperacion transfiere una constante o los contenidos de uno o varios registros a otro registro.

Es importante observar que la transferencia en la microoperacion no es fisicamente instantanea, sino que requiere un tiempo finito. Usualmente se escoge un periodo de reloj mayor que la cantidad de tiempo requerida para la realizacion de la microoperacion. Siempre se debe tener en cuenta que el simbolo (flecha) implica un retardo. En ocasiones este retardo es beneficioso:

D<- inc D

Se distinguen cuatro tipos de microoperaciones:

* De establecimiento de constantes:

G<-1, C<-0, B<-8, D<-9

* Unitarias:

D<-inc D, A<- shr A, B<- B'

* Binarias:

A<- A add R, B<- B sub A

* Solo de transferencia:

Operadores	Símbolos	Explicaciones
Operadores lógicos		
NOT	'	Operación lógica no
OR	+	Operación lógica o
AND	*	Operación lógica y
EXOR	⊕	Operación lógica o exclusivo
COIN	⊙	Operación lógica coincidencia
Operadores funcionales		
Desplazamiento a la izquierda	SHL	Desplazamiento de uno o mas bits a la izquierda
Desplazamiento a la derecha	SHL	Desplazamiento de uno o mas bits a la derecha
Desplazamiento circular a la izquierda	CIL	Desplazamiento circular de uno o mas bits a la izquierda
Desplazamiento circular a la derecha	CIR	Desplazamiento circular de uno o mas bits a la derecha
Incrementar	COUNTUP o INC	Incrementar la cuenta de uno
Decrementar	COUNTDN o DEC	Decrementar la cuenta de uno
Operadores aritmeticos		
Suma	ADD	Sumar un numero binario sin signo a otro
Resta	SUB	Restar un numero binario sin signo de otro

Tab. 1.1.- Operadores basicos admitidos en el CDL.

A<- B, B<- C(ADDR)

1.1.2.3.- MICROOPERACIONES CONDICIONALES.-

Reproduce la estructura condicional de programacion tipica:

IF (<condicion>) THEN (<microsentencias.si>)
ELSE(<microsent.no>)

La seccion ELSE es opcional, y tanto las microsentencias, como la condicion si se encuentran abrazadas por sendos parentesis.

IF (G=0) THEN (F <- 10),

IF (G 1) THEN (C <- 4, d <- 5) ELSE (F <- 11),

Las microsentencias encerradas entre parentesis se realizan simultaneamente, en paralelo, por ello el orden no tiene importancia.

Las condiciones pueden afectar a varios bitios:

IF (C=77) THEN (...)

Dado que las condiciones tienen un valor logico (1=cierto, 0=falso), pueden emplearse operadores logicos:

IF ((X(1)=1)*(Y(2)=0)+(START=ON)) THEN (...)

La expresion condicional puede simplificarse haciendo las sustituciones:

X(1)=1 o X(1)=0 por X(1)

X(0)=0 o X(0)=1 por X(0)'

START=ON por START (ON),

Asi la microsentencia condicional anterior quedara:

IF (x(1)*y(2)+START(ON)) THEN (...)

1.1.2.4.- OPERADORES ESPECIALES.-

La definicion de un operador especial especifica una red logica especial, necesaria para describir algunas microoperaciones.

La definicion del operador especial se hace mediante sentencias CDL, y se le asigna un nombre distinto al de los otros operadores. Un contador Gray de 2 bitios se describira de la forma:

```
Operator,     D <- ctg D(1-0)
/begin/       IF (D=00) THEN  (D <- 01)
              IF (D=01) THEN  (D <- 11)
              IF (D=11) THEN  (D <- 00)
              IF (D=10) THEN  (D <- 00)
              end of operator.
```

1.1.3.- SECUENCIAMIENTO.-

Para realizar una operacion util, se requieren varias microoperaciones. La ejecucion de estas microoperaciones debe ser ordenada y controlada, para lo

cual se añada unas etiquetas a las microsentencias, convirtiendose de esta forma en "Sentencias de ejecucion". Estas estan formadas por una etiqueta, que especifica las condiciones en las que se debe ejecutar la sentencia de ejecucion, y por una o varias, microsentencias, que se ejecutaran simultaneamente.

/P(1)/ C<-inc C, D<-dec D

/P(2)/ IF(C=0) THEN (D<-0)

La segunda sentencia del ejemplo es equivalente a:

/P(2)*(C=0)/ D<- 0

Ya que una etiqueta puede estar formada por una expresion logica. La microoperacion de cargar el registro D con la constante 0, se realizara cuando se cumpla que P(2)=1 y C=0.

Existen basicamente dos tipos de secuencia minuto de un circuito, y un tercero que es combinacion de ambos.

1.1.3.1.- POR RELOJ DE FASES MULTIPLES.-

La sucesion de las fases del reloj es la que determina el orden en que seran ejecutadas las sentencias. Es el metodo empleado en muchos ordenadores para controlar las distintas fases de ejecucion de una instruccion.

```
       Clock, P(3-0)
/P(0)/ RI<-M(CP)
/P(1)/ IF (RI(INDIRECC)) THEN (RI(DIRECC)<-M(RI(DIRECC)))
/p(2)/ IF (RI(CODOP)) THEN (A<-M(DIRECC))
                      ELSE (M(RI(DIRECC))<-A)
/p(3)/ CP<-inc CP
```

1.1.3.2.- POR REGISTRO DE CONTROL.-

Es el metodo usualmente empleado en los pequeños sistemas secuenciales, la etiqueta de cada etapa esta formada por la salida de un decodificador, cuya entrada es modificada por una microoperacion de cada sentencia.

```
          Clock, P
          Decoder, D (0-3) = E (1-0)
/D (0)*P/  A<-B, E  inc E
/D (1)*P/  B<-C, E<-inc E
/D (2)*P/  C<-A, E<-0
```

1.1.3.3.- MIXTO.-

Este metodo resulta combinando los dos metodos anteriores, se emplea en los ordenadores para describir la fase de ejecucion de las distintas instrucciones.

```
          Decoder, D(0-10)=RI(CODOP)
          Clock, P(2-0)
```

```
$ Ejecucion de la instruccion suma.CODOP=1
/P(1)*D(1)/     B<-M(RI(DIRECC))
/P(2)*D(1)/     A<-A add B,  CP<-inc CP
$ Ejecucion de la instruccion resta. CODOP=2
/P(1)*D(2)/     B<-M(RI(DIRECC)
/P(2)*D(2)/     A<-A sub B, CP<-inc CP
$ Ejecucion de la instruccion de salto. CODOP=8
/P(1)*D(8)/
/P(2)*D(8)/     CP<-RI(DIRECC)
```

1.2.- KARL.-

Es un lenguaje propuesto por R.W.Hartenstein en el libro. "Fundamentals of Structured Hardware Design" (1977). Esta basado en la transferencia de registros, y en el CDL, al que mejora. Permite describir con facilidad circuitos digitales, tanto sincronos como asincronos, partiendo de descripciones algoritmicas del tipo de las redes de Petri.

Paralelamente propone en "lenguaje" para descripcion grafica de los circuitos, al que llama ABL. El hecho de sugerir los dos metodos de descripcion en paralelo refleja uno de los objetivos de este lenguaje: aproximar la descripcion a la realizacion final mediante circuitos integrados. El lenguaje ha sido definido con mucho mas rigor y permite la realizacion de diseños modulares y estructurados.

Tal como hemos dicho al principio, en su descripcion haremos referencia al CDL, puesto que este lenguaje pretende superarlo.

1.2.1.- TIPOS DE INFORMACION.-

Partiendo de la base dse que se trabaja con informacion binaria, se distinguen varios tipos de codificacion de la informacion.

1.2.1.1.- NUMERICA.-

Se considera asi cuando a cada combinacion posible se le asocia un valor numerico. Este lenguaje permite definir cuatro tipos de variables numericas:

tipo	palabra clave	descripcion	ejemplo
Entera	int	= di B (B=8.16.10)	10 10
Binaria	binary	Entero con B-2	1010
Unaria	unary	Entero con B=1	0011.1111
Singular	singulary	Uno entre n	0100.0000
singular multiple	multiple singulary	(uno entre n) (3,5,7)	=1010.1000

1.2.1.2.- NO NUMERICA.-

Se considera asi a la informacion, cuando la combinacion de unos y ceros que la representa no tiene asignado ningun valor. Se puede codificar de dos formas:

tipo	palabra clave	Descripcion
Booleana	BOOLEAN	Un solo bitio, el cual puede valer verdadero o falso
Logica	LOGICAL	n bitios, no tienen valor.

1.2.2.- CLASES DE SENTENCIAS.-

Basicamente se consideran los mismos tipos de sentencias que en el CDL, segun que describan las unidades de almacenamiento de datos o una transferencia estatica o dinamica.

1.2.2.1.- SENTENCIAS DECLARATORIAS.-

Son las que definen los elementos de almacenamiento de los datos. Admite los siguientes tipos:

Registro	REGISTER
Sub registro	SUBREGISTER
Matriz de registros	ARRAY-REGISTER
Sub matriz de registros	SUB ARRAY-REGISTER
Memoria	MEMORY
Cascada	CASREGISTER
Constante	CONSTANT
Matriz constante	ARRAY-CONSTANT

1.2.2.2.- SENTENCIAS APLICATIVAS.-

Son las empleadas para describir las conexiones incondicionales del circuito, es decir, aquellas que no dependen de ninguna condicion dinamica. Tambien permiten describir los operadores combinatorios del circuito.

Estas sentencias admiten varias versiones:

Terminal	TERMINAL
Terminal en cascada	CASTERMINAL
Sub-terminal	SUBTERMINAL
Codificador	ENCODER
Matriz-terminal	ARRAY TERMINAL
Submatriz-terminal	SUBARRAY TERMINAL
Multiplexor	MULTIPLEXOR
Demultiplexor	DEMULTIPLEXOR

En la tabla 1.2. se decriben los distintos operadores admitidos por este lenguaje. Cabe destacar el completo juego de operadores del desplazamiento, de prioridad y de descripcion de impulsos.

1.2.2.2.1.- OPERADORES ARITMETICO/LOGICOS.-

Se distinguen dos tipos:

Unarios: TERMINAL H=not Q

TERMINAL FG=(Q=3) ; operacion de reconocimiento

Binarios: TERMINAL b=Q-ALO, b=Q+ALO

TERMINAL FF=(Q=ALO), FF=(Q<=ALO)

FF y F son variables Booleanas, y valen verdadero o falso, segun sea el resultado de la operacion relacional, indicada entre parentesis.

1.2.2.2.2.- OPERADORES DE DESPLAZAMIENTO.-

Simbolos	palabras llave	Explicaciones
Operadores logicos		
¬	not	NOT	Inversion logica
∧	and	AND	And logica
∨	or	OR	Or logica
⊻	exor	EXOR	Excl.or logica
≡	coin	COIN.	coincidencia o entidad logica
→	impl	IMPL	implicacion
Operadores aritmeticos		
*	<x>	MUL	multiplicacion
/	</>	DIV	division
	mod		moculo
+	+	PLUS	suma
-	-	MINUS	resta
+1	<+1>	INC	incremento
-1	<-1>	DEC	decremento
Operadores de relacion		
=	=	EQ	igual
=	=/	NE	desigual o distinto
<	<	LT	menor que
>	>	GT	mayor que
≤	=<	LE	menor o igual que
≥	>=	GE	mayor o igual que
Operadores funcionales		
	SHL	despl. izq. (no destruye)
	SHR	despl. der. (no destruye)
	CIL	despl. circular izq.
	CIR	despl. circular der.
	DSHL	despl. destruct. izq.
	DSHR	despl. destruct. der.
	CSHL	despl. constructivo izq.
	CSHR	despl. constructivo der.
	CSHL	despl. aritm. extendido izq.
	PUSH	movimiento destruct. izq.
	POP	movimiento destruct. der.
	NPUSH	movimiento no destruct. izq.
	NPOP	movimiento no destruct. der.
	REFL	reflejar o reflexion
	XFER	conexion.
Operadores asignacion:		
:=	:=	<:=>	asignacion sincrona
=	=	<=>	conexion
<=	<=	xfer	"
=	=	=	conexion bidireccional
<=>	<=>	bid	" "

Tab. 1.2.- Operadores basicos aritmeticos en el LAMI.

Operadores de prioridad:

>	.>	PRL	Prioridad a la izquierda salida singular
<	<.	PRR	Prioridad a la derecha salida singular
o>	->	NEG-PRL	Prioridad a la izquierda salida singular negada
<o	<-	NEG-PRR	Prioridad a la derecha salida singular negada
⊃	.)	UPRL	Prioridad a la izquierda salida codif. unaria
⊂	(.	UPRR	Prioridad a la derecha salida codif. unaria
⊃	-)	NEG-UPRL	Prioridad a la izquierda salida unaria negada
⊂	(-	NEG-UPRR	Prioridad a la derecha salida unaria negada

Otros:

unary	unary	UNARY	indicador de variable unaria
∪	conv	CONV	convergencia
:	:	TO	rango
sing	sing	SING	indicador de variable singular
		< >	blanco
:	:	CAS	cascada (operandos)
:	:	CAT	encadenamiento (operadores)
:	:	ASSOC	asociador (reduccion)
:	:	REPL	replicador
:	:	<:>	separador de etiqueta
mut-sing	mult-sing	MULT-SING	indicador de var. singular multiple.

Terminales de clausulas:

IF...	THEN...	ELSE...	FI	condicion
WILE...	KEEP...	ELSE...	ELIW	asignacion asincrona
AT...	DO...	TA		asignacion asincrona por flanco
ON...	DO...	NO		asignacion asincrona amo/esclavo
AT...	TIL...	KEEP...	ELSE...TA	asignacion asincrona, inici/paso
CASE...	OF...	ELSE...	ESAC	clausula case.

Tab. 1.2.- (continuacion).

En los desplazamientos, el primer bit de la cadena puede permanecer invariante (no destructivo), o ser puesto a cero (destructivo). El ultimo bit de la cadena se pierde, salvo en el caso del cshl, en que permanece inalterado para conservar el signo y el que se pierde es el penultimo.

En los desplazamientos circulares el primer bit toma el valor del ultimo, salvo si el desplazamiento es constructivo, en cuyo caso se pone a 1.

1.2.2.2.3.- OPERADORES DE RETARDO Y GENERACION DE PULSOS.-

Permiten describir un retardo constante o variable, dentro de ciertos margenes. Este retardo puede ser igual o distinto para los flancos ascendente y descendente.

El operador DIFF genera un pulso a partir del flanco ascendente del operando.

En la fig. 1.1. se muestran las posibilidades de estos operadores.

1.2.2.3.- SENTENCIAS DE TRANSFERENCIA IMPERATIVA.-

Son aquellas que se realizan en el momento de producirse la señal de control, esta señal de control puede ser condicional continua o estatica, y discreta,

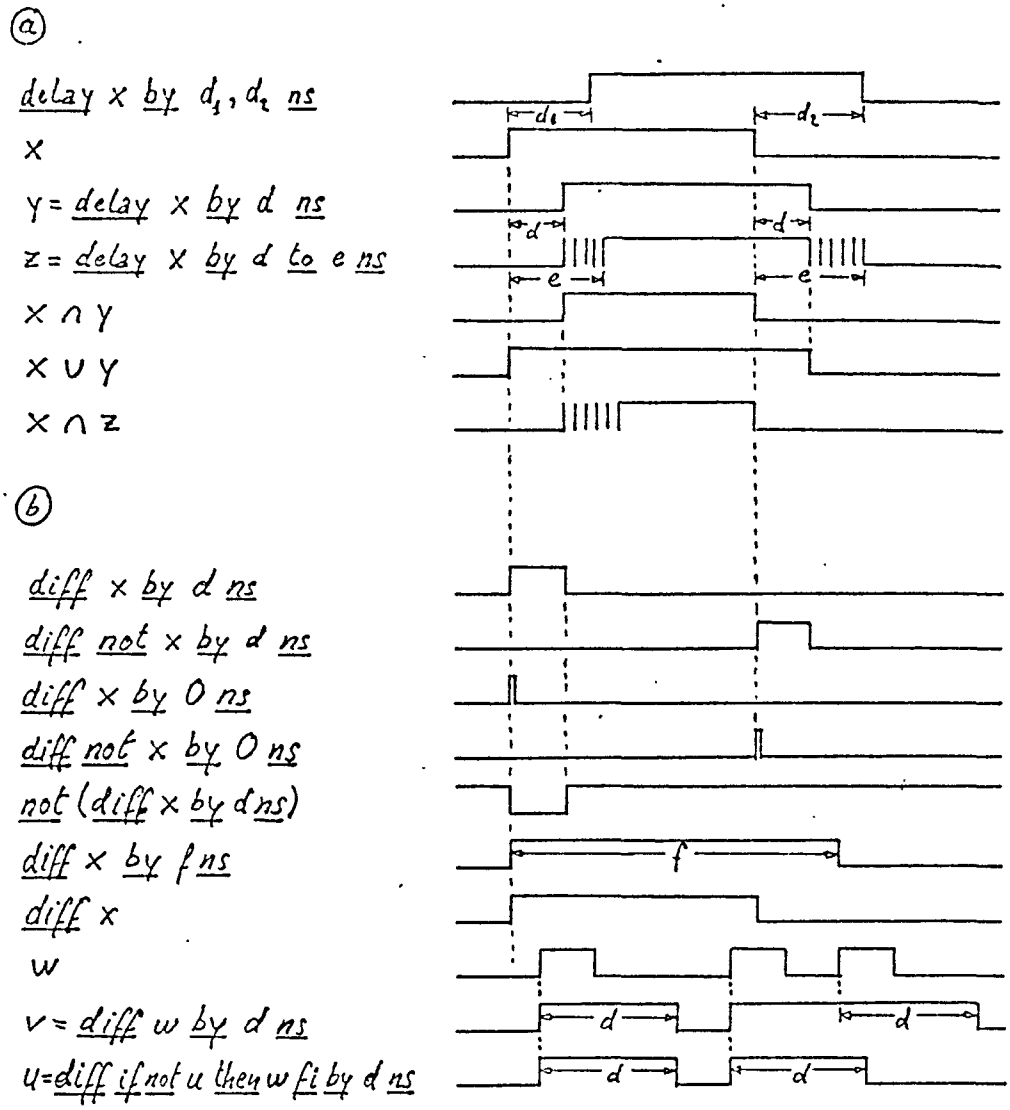


Fig. 1.1.- Operadores temporales del VARI:
a) Operadores de retardo. La señal x sirve de referencia para el calculo de las otras. Obsérvese la posibilidad de definir un retardo variable respecto a los flancos ascendente y descendente de una señal (Z).
b) Operadores de pulsos. Las dos ultimas señales muestran la forma de definir un monostable con estos operadores, segun sea redisparable (V) o no (U).

imperativa o dinamica.

La fig. 1.2. muestra los tres tipos de señales control consideradas por el lenguaje:

Flanco ; at __ do _ ta

Pulso ; on __ do _ no

Nivel ; wile _ keep _ _ else _ eliw

1.2.2.4.- SENTENCIAS DE PERIFERICOS.-

Son las que permiten describir los terminales de comunicacion con el entorno del circuito. Son igual que en el CDL, salvo que admiten unos operadores para cambio del tipo de informacion binaria o singular y viceversa. Asi se puede construir las sentencias:

```
LIGHT       SINGULARY  CREG(15:0)=OUTREG (3·0)
SWITCH      D (3:0)=SINGULARY (9:0)
```

El KARL, admite ademas otra forma de declaracion de variables y funciones externas, mediante la sentencia:

```
UNIT (arg.entr.1 arg.entr.2....,var.sal 1,var.sal2)
        declaraciones
        sentencias
TINU
```

Esta sentencia introduce el concepto de modulo, y por tanto permite describir un bloque del circuito de forma aislada, y referenciarlo cuando sea necesario con los - argumentos actuales.

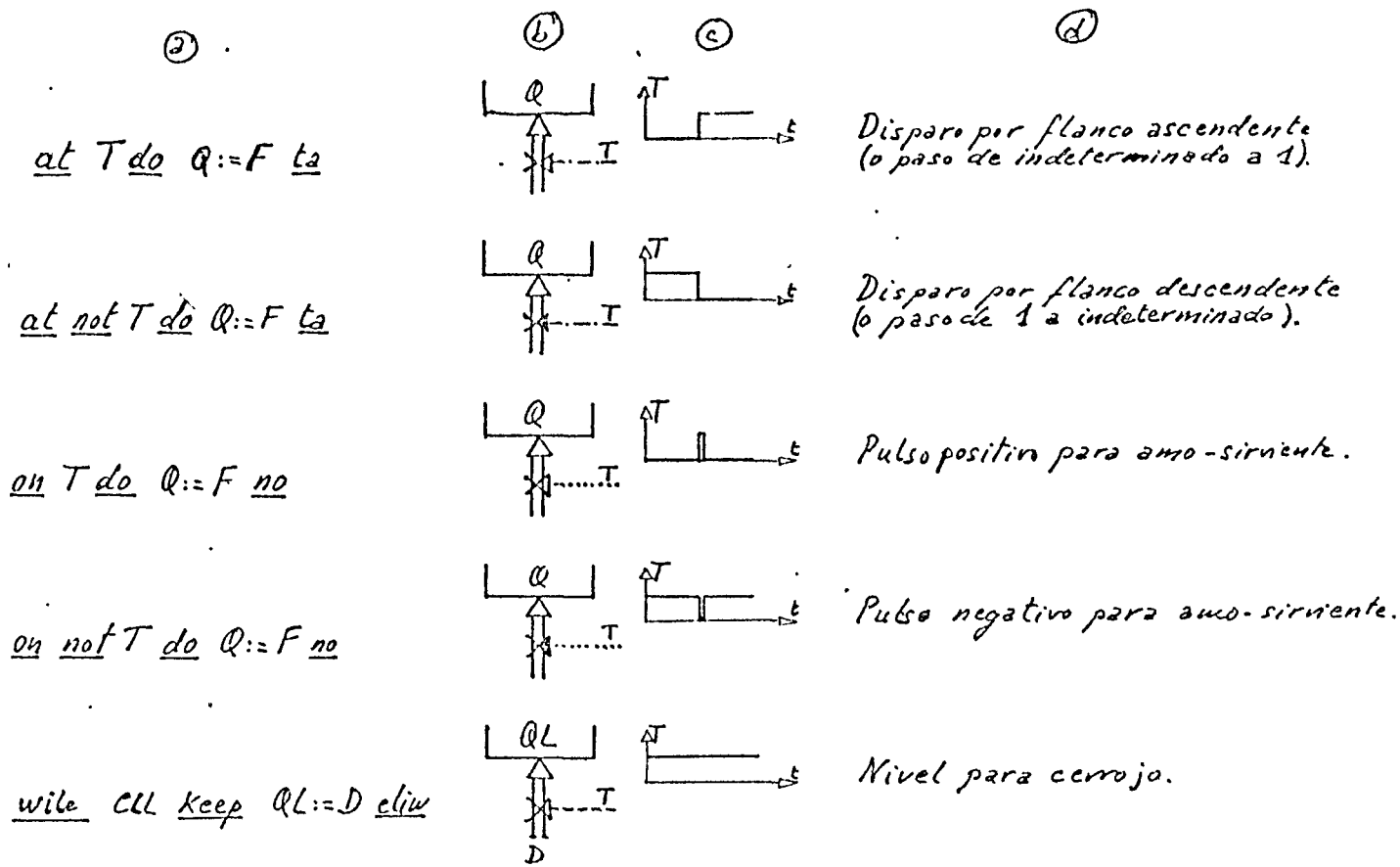


Fig. 1.2.1.- Distintas representaciones de las señales de control para las transferencias imperativas del API:

a) Secuencial (KARL)
b) Diagrama de bloques (API)
c) Temporal (cronograma)
d) Descripcion.

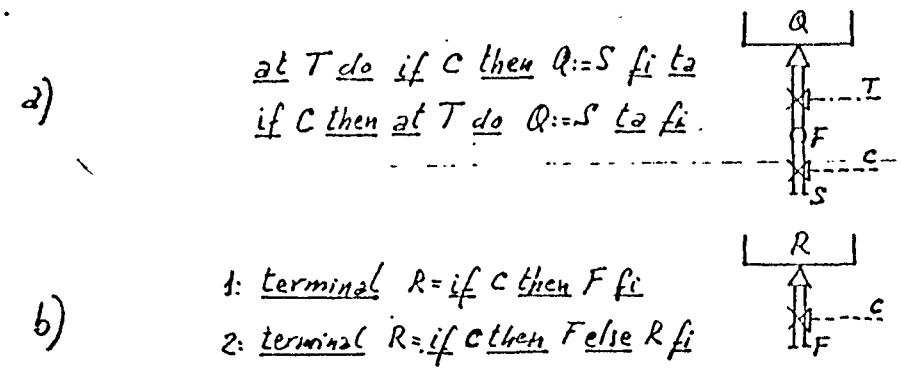


Fig.1.2.2.- Ejemplos de descripciones equivalentes en forma secuencial (YARI) y en diagrama de bloques:
a) Señal de reloj con validacion.
b) Condicion condicional:
1: Sin almacenamiento. R no esta definida si C no esta activa.
2: Con almacenamiento. R esta definida siempre (cerrojo).

Esta es tal vez la principal ventaja de este lenguaje frente al CDL, a parte de la mayor flexibilidad de los operandos y datos.

Se puede definir, por ejemplo, una pastilla integrada cuadruple multiplexor de 2 canales:

```
UNIT  TTL 98(PE1(3:0).PE2(3:0), AE, TRIGGER T; PA(3:0)
   REGISTER PA(3:0);
   AT T DO PA := IF AE THEN PE2 ELSE PE1 TA
TINU
```

1.2.2.5.- RELOJ.-

Se pueden definir relojes mono o multifasicos, como en el CDL:

```
Clock, P(3:0)
```

1.2.3.- ESTRUCTURAS DE CONTROL.-

En este lenguaje existen dos mecanismos para controlar la ejecucion de una operacion, las sentencias imperativas, que ya hemos descrito, y las clausulas condicionales o bien una mezcla de ambas.

1.2.3.1.- CLAUSULAS CONDICIONALES.-

Son las tipicas de todos los lenguajes:

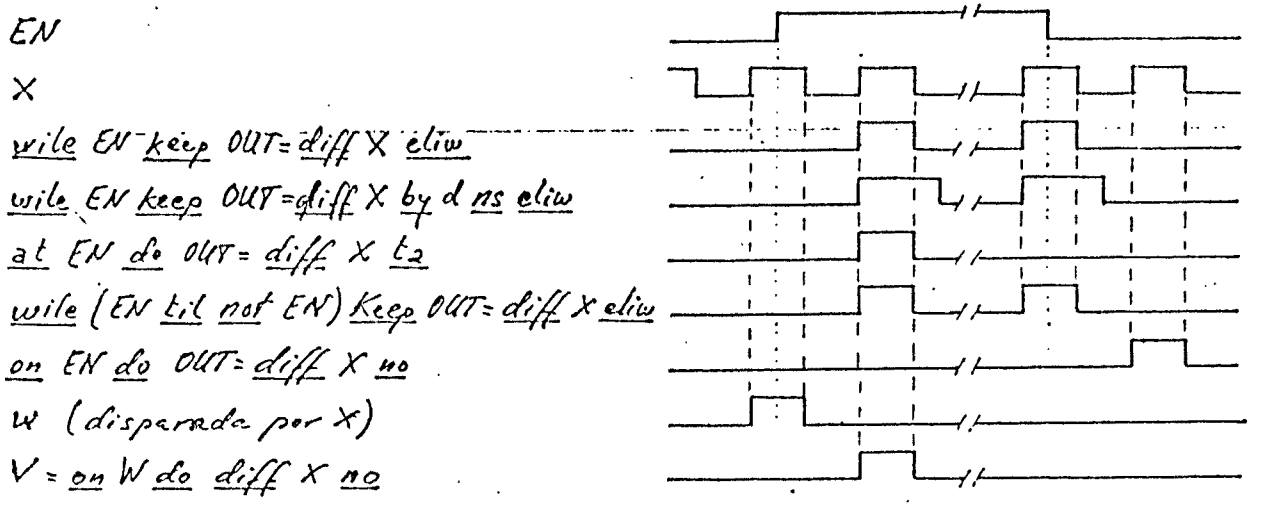


Fig. 1.3.- Señales de control compuestas. La ultima señal (V) constituye un ejemplo de "retardo controlado"

IF <cond> THEN <sent-si> ELSE <sent-no> FI

CASE <expresion> OF (<lista const.i>:<sent.i>)

ELSE <sent.no>

ESAC

La clausula CASE supone una mejora del CDL pues permitedescribir la ejecucion condicional de sentencias de una forma mas clara que con un decodificador.

1.2.3.2.- ESTRUCTURA MIXTA.-

En general las señales de control estaticas se reflejan en el lenguaje mediante las clausulas condicionales. Estas clausulas, lo mismo que las transferencias incondicionales son validadas en la secuencia correcta mediante las sentencias imperativas.

La fig. 1.3. muestra los efectos de los tres tipos de señales de control, para la generacion de pulsos.

La fig. 1.4. muestra la equivalencia entre las descripciones grafica y secuencial de un registro de desplazamiento de la serie TTL convencional.

```
CDL)                Clock CK
                    Switch, EP(7_0),ES,CAR,VAL
                    Light, SS(1_0)
                    Register, R(7_0)
                    Decoder, SS(1_0)= R(0)
           /1/      IF  CAR<-THEN  R<-EP
           /CK*VAL/ IF  CAR'  THEN  R<-ES_R(7_1)
```

```
KARL)      UNIT TTL165 (EP(7:0), ES,VAL,TRIGGER (CK),CAR,L;SS(1:0));
           REGISTER R(7:0);
           WILE CAR  KEEP  R= EP ELSE
                AT CK DO IF VAL THEN ( ): R= SHR ES: R FI TA ELIW
           TINU
```

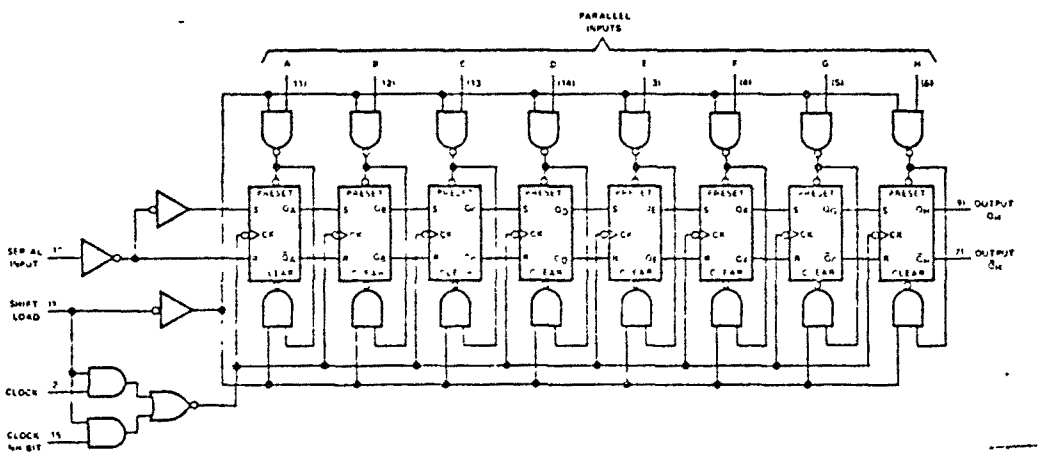


Fig. 1.4.- Descripcion del registro de desplazamiento 74165 en distintos lenguajes:
a) CDL
b) KARI
C) Logigrama convencional.

RESUMEN DEL CAPITULO:

Como conclusion de todo lo dicho podemos extraer las caracteristicas que debe tener un lenguaje de transferencia de registros LTR aplicable a la descripion de sistemas basados en microprocesador. Debe permitir la descripcion de los tres aspectos que comprende todo circuito:

- Elementos de almacenamiento y terminales.

- Operadores combinatorios entre datos.

- Secuencias de microoperaciones.

El nivel de detalle con que se especifiquen estas secuencias de microoperaciones debe poderse adaptar a las necesidades de cada caso. En concreto interesan dos niveles. Uno en el que se suponga la existencia de un secuenciador controlado por contador, al estilo del contador de programa de un procesador, y que permita una descripcion sin especificar las condiciones de ejecucion, obteniendo asi una version simplificada del CDL, en la que se detalla escuetamente la informacion imprescindible para comprender el funcionamiento del circuito. El otro nivel debe permitir una descripcion a nivel de control de transferencias de informacion por los buses del procesador, añadiendo a la descripcion anterior las primitivas de control del KARL, que permiten especificar microoperaciones condicionadas por flancos de señales de

control, tal como exigen las normas de sincronismo de los buses de la mayoria de los microprocesadores.

Dado que en la mayoria de los casos, el diseñador de microprocesadores, no va a contar con un rapido y potente ordenador capaz de soportar un simulador, se ha pensado que era mucho mas interesante proponer el uso de un lenguaje que, frente a la desventaja de no ser simulable, presiente la enorme ventaja de ser flexible, gramaticalmente sencillo y perfectamente valido para la documentacion y descripcion formal de un circuito. De esta forma se elimina el factor que ha restado difusion a los lenguajes descriptores de circuitos entre los diseñadores de pequeños sistemas potenciando el lenguaje mas que como util de diseño, como metodo de descripcion del funcionamiento, imprescindible cuando el diseño se realiza por mas de una persona trabajando en equipo.

2.- HERRAMIENTAS PARA EL DISEÑO DE PROGRAMAS.

INTRODUCCION.-

Ya se ha comentado la falta de formacion de los diseñadores de sistemas basados en microprocesador en el campo de la programacion, pues hasta ahora la mayoria de ellos provenian del campo del diseño de circuitos, empleando los microprocesadores como un componente mas. Tambien se ha dicho que la creciente complicacion de los sistemas a diseñar hace imprescindible la importacion al terreno de los microprocesadores de las tecnicas de programacion mas avanzadas. Por esta razon se ha incluido en este capitulo una breve recopilacion de dichas tecnicas, haciendo como siempre especial referencia a las caracteristicas mas provechosas para el diseñador de sistemas basados en microprocesador.

Primeramente se han expuesto las bases de la programacion estructurada, pues nadie duda de la necesidad de aplicar a la programacion los conceptos que engloba el termino, aunque se empiece a cuestionar el empleo del termino en si, por considerar evidente que toda programacion, para serlo realmente, debe ser estructurada.

A continuacion se describen las tecnicas mas empleadas en la concepcion de programas, o mejor dicho de los algoritmos que van a ejecutar dichos programas. De las dos metodologias expuestas (HIPO y JACKSON), se deduciran

las caracteristicas que debe tener la metodologia a aplicar en la concepcion de SBMP.

Finalmente se analizan los lenguajes empleados en la descripcion y documentacion de los algoritmos, pues ya se ha dicho que uno de los objetivos del trabajo es generar una documentacion clara y asequible del diseño. Se han incluido dos lenguajes secuenciales PDL y PASCAL, que a pesar de ser combinables, y por tanto directamente ejecutables, proporcionan una descripcion del algoritmo bien estructurada y de facil interpretacion para un tecnico ajeno al diseño. Tambien se incluye finalmente, en lenguaje grafico para la descripcion de algoritmos en forma de diagramas, aunque el metodo de concepcion de JACKSON incluye unos diagramas en la descripcion, estos son demasiado generales para detallar algunas caracteristicas peculiares de la programacion en lenguajes de bajo nivel o el diseño de circuitos. Por otra parte, los diagramas de Nassi-Schneidermann constituyen una alternativa a los tradicionales diagramas de flujo, ciñendolos a las limitaciones de la programacion estructurada.

2.1.- PROGRAMACION ESTRUCTURADA.-

2.1.1.- CARACTERISTICAS GENERALES.

Los metodos propuestos por los "padres" de la programacion estructurada: Dijkstra, Hoare, Wirth, Jackson, etc., se pueden dividir en dos grandes grupos:

* Metodos orientados al analisis de los problemas

* Metodos orientados a la sistesis de los programas.

Ello refleja una de las caracteristicas fundamentales de la estructuracion, y es la conveniencia de analizar a fondo las hipotesis del problema como paso previo al diseño de los programas. Esta es la base de todos los metodos llamados descendentes (top-down).

Otro de los principios fundamentales de la programamcion estructurada es el uso de un numero limitado de estructuras en la confeccion de los programas. Ello se contrapone a los metodos basados en los organigramas convencionales en los que la ausencia de estructuras rigidas, permitia al programador construir las estructuras mas imprevisibles. Precisamente la rigidez de los metodos estructurados, unida a la relativa incompatibilidad con lenguajes como el FORTRAN, han sido las causantes de las dificultades encontradas para la difusion de los metodos estructurados.

2.1.2.- VENTAJAS.-

Hay mucha literatura escrita sobre la forma de diseñar un programa. Toda ella tiene una cosa en comun, y es la defensa de algun metodo de programacion estructurada. Unos defienden un metodo, otros otro, pero en el fondo todo el mundo esta de acuerdo en que el uso de algun metodo estructurado reporta indudables ventajas frente a los "metodos" que confian el exito del diseño a la inventiva o "feliz idea" del programador o analista.

Si seguimos extrapolando los distintos procedimientos de diseño estructurado, veremos que, junto a una mejor documentacion y legibilidad de los programas se obtiene un aumento del rendimiento de los programadores del orden del 100%. La Tabla 2.1. muestra algunos datos significativos, obtenidos de dos proyectos de la NASA.

Ahora vamos a describir brevemente algunos de los metodos de programacion estructurada mas interesantes.

APOLO	METODOLOGIA	Noctetos	Mes/Hombre	Octeto/Mes*Hombre
MISION/OPERACIONES/CONTROL	DLS	5.8	3748	1547
TIERRA/SOPORTE/SIMULACION	NINGUNA	2.1	1809	1161

SKYLAB	METODOLOGIA	Noctetos	Mes/Hombre	Octeto/Mes*Hombre
MISION/OPERACIONES/CONTROL	DLS	1.4	1665	841
TIERRA/SOPORTE/SIMULACION	DLS/PE/DD	4.0	1065	3756

Tab. 2.1.- Estadisticas de la NASA mostrando los rendimientos obtenidos empleando distintas metodologias de programacion:
DLS: Desarrollo de Librerias de Soporte.
PE: Programacion Estructurada.
DD: Desarrollo Descendente.

2.2.- HIPO: HIERARCHY AND INPUT-PROCESS-OUTPUT.-

Es uno de los primeros metodos de programacion desarrollados, y se basa en el analisis descendente de la entrada, los procesos y la salida del sistema a diseñar. Data de mediada la decada de los anos 70, y su promotor es J.F.STAY / /. Actualmente esta superado por otros metodos, pero el hecho de que sea utilizado por IBM le mantiene de actualidad.

2.2.1.- FASES DEL DESARROLLO DE UN PROGRAMA.-

* Definicion de los requisitos.

* Analisis del sistema.

* Diseño del sistema.

* Diseño del programa.

* Diseño detallado de los modulos.

* Documentacion del programa y del sistema.

2.2.2.- HERRAMIENTAS DE ANALISIS Y SINTESIS.-

Para el analisis del sistema se emplean unos diagramas esquematicos, jerarquizados, en los que se

reflejan las distintas descomposiciones de que va siendo objeto cada una de las partes del programa. (fig. 2.1.a), asi como las relaciones entre ellas.

Para la sintesis y documentacion de los programas, se confeccionan unas tablas como la de la fig. 2.1.b, en las cuales se describen con mayor detalle las funciones o tareas que lleva a cabo cada modulo, asi como las variables que procesa y produce.

2.2.3.- PROCEDIMIENTO MENTAL DE DISEÑO.-

* Identificacion de las funciones a realizar por cada modulo.

* Descomposicion funcional del modulo.

* Diseño iterativo. Esto es, que el proceso de identificacion de funciones y descomposicion funcional se repite hasta llegar a funciones ya resueltas, mediante instrucciones del lenguaje o mediante rutinas de la libreria.

* Interrelaciones entre modulos. Influencias de un modulo en sus subordinadas y en sus superiores jerarquicos.

* Optimizacion retardada del rendimiento del programa. De esta forma se evita la generacion de un

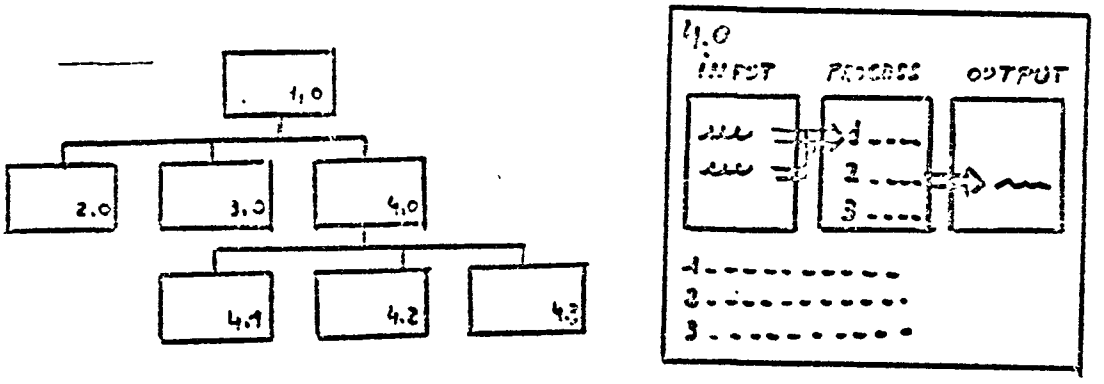


Fig. 2.1.- Estructuracion de la informacion en el metodo HIPO:
a) Diagramas jerarquizados. Muestran los procesos en que se descompone cada tarea.
b) Tablas modulares. Formulario en el que se describen los procesos a los cuales son sometidas las variables de entrada para obtener los resultados de salida.

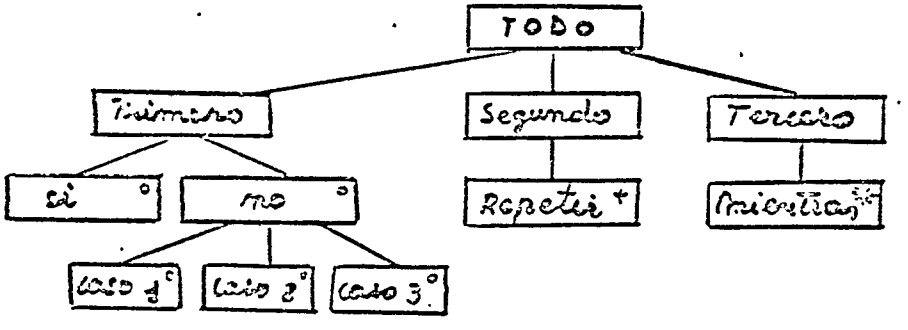


Fig. 2.2.- Estructuras de datos y control representables mediante los diagramas de Jackson:
Todo: Secuencia
Primero: Cordicional o Seleccion
Segundo: Iteracion, al menos una vuelta.
Tercero: Iteracion, tal vez ninguna vuelta.

codigo excesivamente simplificado, que dificulta su interpretacion, al estar muy lejos de las estructuras en que se ha descompuesto el problema.

2.2.4.- BENEFICIOS OBTENIDOS.-

* Facil comprension del contenido funcional.

* Pronta identificacion de la informacion inconsistente u omitida.

* Las funciones son discretas y en consecuencia mas facilmente documentables y modificables, si conviniera.

* La documentacion se obtiene con un unico esfuerzo en vez de repartirlo en distintas etapas del diseño.

* Los interfaces entre modulos son simples, y por tanto se reduce la probabilidad de errores logicos.

* El diseño resultante conduce a un codigo descendente.

* El mantenimiento y las mejoras son facilmente transferibles, pues el sistema es comprensible a todo el mundo.

2.3.- METODOLOGIA JACKSON.-

Tambien data de mediados de los años 70. Su nombre proviene del de su promotor, Michael Jackson / /. Esta orientada al analisis de los datos de entrada y de salida, y a partir de este sintetiza los programas.

2.3.1.- PROCEDIMIENTO DE DISEÑO.-

2.3.1.1.- El procedimiento de diseño se puede resumir en tres puntos:

* Definicion de las estructuras de datos.

* Creacion de la estructura del programa a partir de ellas.

* Expresar las tareas del programa en instrucciones ejecutables.

Cuando el programa cuenta con mas de una estructura de datos (entrada y salida tipicamente) debe ajustarse a todas ellas, tomando elementos de una y otra estructuras. Generalmente se podra establecer una relacion entre los distintos elementos de cada estructura de datos.

2.3.1.2.- Los objetivos que animan cada una de las tres

fases del diseño descritas son las siguientes:

* Refinamientos sucesivos de las descripciones.

* Descomposicion funcional de las tareas.

* Programacion por conjuntos de actuacion.

2.3.1.3.- Concretando mas, el metodo contempla los siguientes aspectos:

* Modelizacion del entorno del problema en estructuras de datos jerarquicas.

* Expresion de la tarea a realizar por el programa, en operaciones ejecutables elementales. Dichas operaciones actuaran sobre componentes elementales de los datos.

* Ademas de estas operaciones primarias, se requieren operaciones secundarias como leer o escribir.

* Cada operacion se coloca en el lugar adecuado del programa.

* Las estructuras que se emplean, se corresponden con las de los datos.

* Las operaciones ejecutables, primarias y secundarias, se ajustan perfectamente a esas estructuras.

2.3.2.- ESTRUCTURAS DE FLUJO.-

Se reconocen tan solo tres estructuras de flujo:

* Secuencia o sucesion de tareas (fig.2.2.a).

* Itearacion, bajo la tipica forma DO/WHILE, en contraposicion a la REPEAT/UNTIL, que no se considera (fig.2.2.b).

* Seleccion, sin distinguir entre dos o mas caminos posibles.

2.3.3.- JUSTIFICACION DEL METODO.-

El metodo puede justificarse de forma general o en cada uno de sus aspectos particulares. A nivel general presenta las siguientes ventajas:

* No es fruto de la inspiracion.

* Es racional. El proceso de diseño se basa en unos principios (los datos), que permiten validar cada paso de la solucion.

* Es didactico, facil de enseñar a la gente. Dos programadores que lo usen, llegaran basicamentea la misma solucion.

* Es practico, facil de comprender. Los diseños producidos se pueden traducir facilmente a cualquier lenguaje de programacion.

Por todas estas razones, este es uno de los metodos de estructuracion de programas mas usados a nivel internacional.

2.4.- DESCRIPCION DE ALGORITMOS.-

Una vez escritos los metodos para la estructuracion de los programas, vamos a ver los metodos mas comunmente empleados para la descripcion detallada de los algoritmos que realizaran esos programas.

Una vez desdoblado el problema global en una serie de programas elementales, ligados entre si mediante una cierta estructura, para cada uno de estos programas procede considerar las siguientes fases:

* Definir todos los interfaces externos e internos.

* Definir todas las situaciones de error.

* Identificar todos los procedimientos que comprende el programa.

* Identificar todas las llamadas a procedimientos subordinados.

* Definir todos los datos globales.

* Definir todos los bloques de control.

* Especificar los algoritmos de todos los procedimientos.

CONVENIO DNF EMPLEADO:

{A}::= Puede repetirse A, entre 0 y n veces
[A]::= Puede darse A opcionalmente, maximo 1 vez.

a) IF <condicion_1>
   <sentencias_si>
   {ELSEIF <cond_2>
   <sentencias_no_si}
   [ELSE <sentencias_no>]
   ENDIF

b) DO CASE OF <criterio_de_seleccion>
   {<caso_i>:<sentencias_i>}
   [OTHER:<sentencias_no>]
   ENDDO

c) DO <criterio_iteracion>
   <sentencias_bucle>
   ENDDO

d) <criterio_iteracion>::= WHILE <condicion>
   | UNTIL <condicion>
   | FOR <var>=<exp.1> TO <exp.2> STEP <EXP.3>

e) <sentencias_bucle>::= {<sentencia>
   ENDO (final prematuro)
   CYCLE } (repeticion ")

Tab. 2.2.- Estructuras de control del PDL.
a) Condicional. Obsérvese la posibilidad de encadenar condiciones.
b) Selectiva. La introduccion opcional del caso generico "otro" evita la enumeracion de todas las alternativas no singulares.
c) Iteracion. Admite las tres formas tipicas de control:
d) Mientras; hasta y contador.
e) Permite salir del bucle (EXIT) o reiniciarlo (CYCLE) antes de completarlo.

2.4.1.- PDL: PROGRAM DESIGN LANGUAGE (LENGUAJE PARA EL DISEÑO DE PROGRAMAS).-

Es un lenguaje propuesto por S.H.CAINE y E.K.GORDON.

Los algoritmos son especificados mediante, basicamente, las mismas estructuras que en los lenguajes orientados a la programacion estructurada, con pequeñas variantes :

2.4.1.1.- CONDICIONAL.-

Admite el encadenado de condiciones:

```
IF <condicion_1>
   THEN <sentencias_si>
  (ELSEIF <cond 2>
       <sent_no_si>)
  (ELSE <sent_no>)
ENDIF
```

2.4.1.2.- SELECTIVA.-

Tiene la forma tipica:

```
DO CASE OF <criterio de seleccion>
          ( <caso-i>: <sentencias-i>)
          ( OTHER: <sentencias-no>)
```

ENDDO

2.4.1.3.- ITERATIVA.-

Solo hay una estructura basica, admitiendo una o varias formas para la especificacion de las condiciones de repeticion o salida del bucle.

La forma basica es:

```
DO <criterio de iteracion>
   <sentencias bucle>
ENDDO
```

El criterio de iteracion puede especificar la condicion que se debe cumplir mientras se ejecuta el bucle, o para que deje de ejecutarse, o simplemente un contador:

```
<criterio de iteracion>: = WHILE <condicion> /
                           UNTIL <condicion> /
          FOR <var>=<exp.1> TO <exp.2> STEP <exp.3>
```

Tal vez la caracteristica mas destacable de este PDL sea que, entre las sentencias de un bucle admite sentencias de repeticion o salida prematuras. Esta posibilidad quita rigidez a las estructuras de bucle, sin que ello suponga una perdida en la claridad de la descripcion del algoritmo, salvo el hecho de que la

salida deja de ser unica, y el bucle puede no ejecutarse completamente cada vez.

<sentencias-bucle> := <sentencia> /
UNDO / (final prematuro)
CYCLE (repeticion prematura)

Esta opcion ha sido recogida por Zilog en su lenguaje de alto nivel PLZ.

2.4.2.- PASCAL.-

Este lenguaje ha sido propuesto por N.WIRTH y tiene dos caracteristicas fundamentales:

* Definicion de tipos de variables del usuario, con posibilidad de especificar el conjunto de datos valido.

* Versiones del compilador concurrentes, que permiten la multiprogramacion.

* Compilacion en dos fases:

* La primera analiza sintacticamente el programa fuente y detecta los errores sintacticos o de mezcla de tipos de variables, y produce un codigo intermedio llamado " P-code" (codigo-P).

* La segunda traduce el codigo-P a lenguaje maquina.

PASCAL

```
a) IF <condicion>
      THEN <bloque_sentencias_si>
      [ELSE <bloque_sentencias_no>]

b) DO CASE <expresion> OF
         {<caso_i>:<bloque_sentencias_i>}
   END

c) REPEAT <sentencia> {;<sentencia>}
   UNTIL <condicion_de_final>

d) WHILE <condicion_de_repeticion> DO
         <bloque_sentencias>

e) FOR <var_contr.>:=<exp.valor inic> <sentido cont.><exp.val.final>
   DO <bloque_sentencias>

f) <sentido cont.>::= TO/DOWNTO

g) <bloque_sentencias>::= BEGIN
                          {<sentencia>;}<sentencia>
                          END
                        | [<sentencia>]
```

Tab. 2.3.- Estructuras de control del PASCAL:

a) Condicional
b) Selectiva
c) Repetir la iteracion hasta que se deje de cumplir la condicion.
d) Iterar mientras se cumpla la condicion.
e) Iteracion controlada por contador.

En algunos casos existe la posibilidad de interpretar el codigo-P directamente.

Esta separacion de las fases de compilacion hace relativamente sencilla la construccion de compiladores para nuevas maquinas, pues solo hay que modificar la segunda fase. Tambien se pueden cambiar con facilidad las "palabras clave" del lenguaje, adaptandolo de esta forma al idioma del programador, con lo cual se hace mas didactico para los noveles.

Aunque no se cuente con un compilador de PASCAL en castellano, siempre se puede utilizar el PASCAL-natural o PASCUAL sin las restricciones sintacticas del compilador, para describir los algoritmos de los programas, con mayor flexibilidad.

Las esrtructuras de control del algoritmo utilizadas en el PASCAL son las siguientes:

2.4.2.1.- SENTENCIAS CONDICIONALES.-

Tienen la forma basica:

```
IF <condicion>
   THEN <bloque de sentencias si>
   (ELSE <bloque de sentencias no>)
```

La <condicion> es una expresion de tipo booleano. La

estructura no necesita admitir el anidamiento de condiciones, pues dentro del bloque de sentencias puede incluirse cualquier conjunto de sentencias:

```
<bloque_de_sentencias> : = (BEGIN
                              (<sentencias>.)<sentencia>
                            END /
                            (<sentencias>))
```

2.4.2.2.- SENTENCIA SELECTIVA.-

Tiene la forma:

```
DO CASE <expresion> OF
   (<caso_i>: <bloque_de_sentencias_i>)
END
```

La condicion de ejecucion de cada caso. Es una lista de constantes <caso_i>. Entre todas las listas se deben cubrir todos los valores posibles de <expresion>.

2.4.2.3.- SENTENCIAS ITERATIVAS.-

Admite dos variantes, dependiendo de que el bucle deba o no repetirse al menos una vez. Si puede no ejecutarse nunca se usa:

```
WHILE <condicion_de_ejecucion> DO
      <bloque de sentencias>
```

Si el bucle se debe ejecutar al menos una vez se usa:

```
REPEAT
      <sentencias>  (; <sentencia>)
UNTIL <condicion_de_salida>
```

Ademas, tambien permite especificar un contador para condicionar el final de la ejecucion del bucle:

```
FOR <variable_de_control> := <valor inicial>
                                (TO/DOWNTO)<valor final>
DO <bloque_de_sentencias>
```

2.4.3.- DIAGRAMAS DE NASSI-SCHNEIDERMAN.-

Tambien datan como todos los anteriores lenguajes, de mediada la decada de los 70. Constituyen una alternativa a los organigramas o diagramas de flujo. La ventaja de su uso radica en que solo permite descubrir las estructuras tipicas de la programacion estructurada, uniendo de esta forma la claridad de la grafica a la de la estructuracion. En la fig. 2.3 se muestra la forma que toman las distintas estructuras.

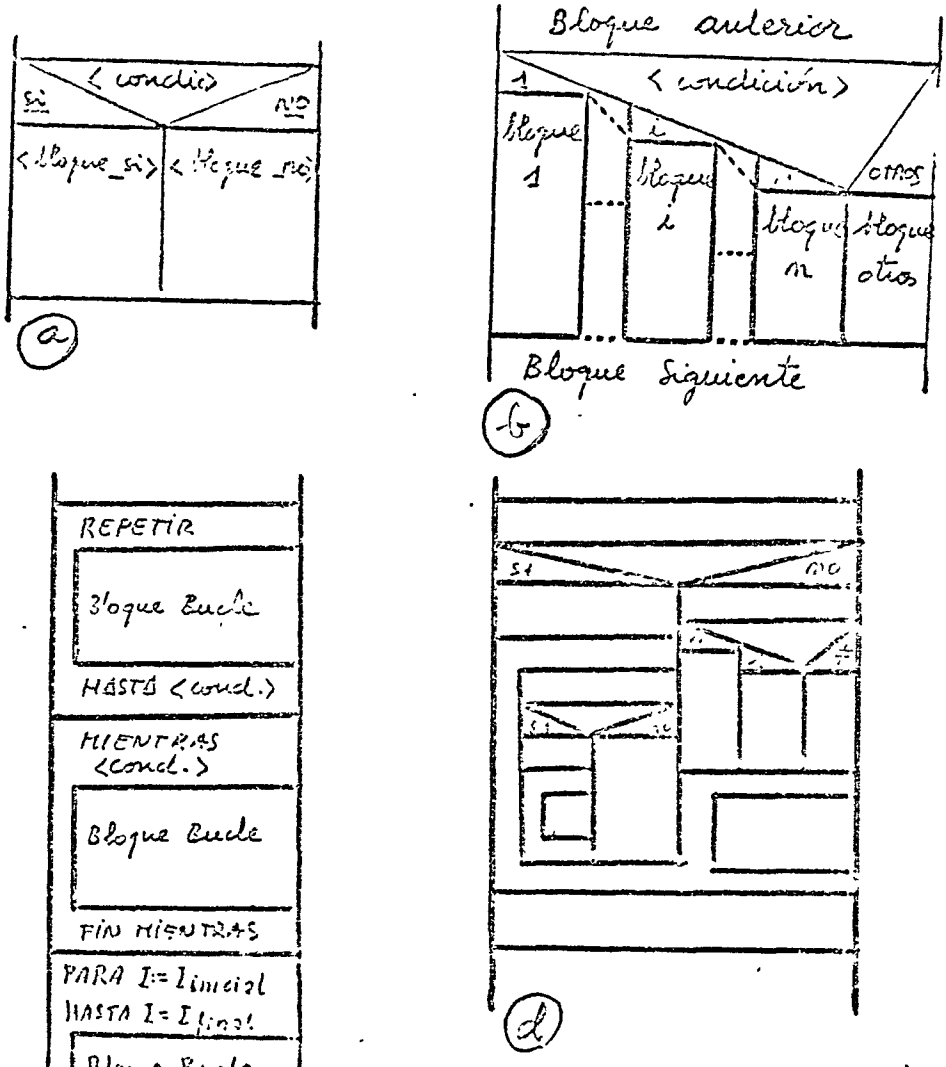


Fig. 2.3.- Diagramas de Nassi-Schneiderrann para la representacion grafica de algoritmos. Admite las estructuras:
a) Condicional.
b) Selectiva.
c) Iterativa.
d) Ejemplo de combinacion de varias estructuras.

RESUMEN DEL CAPITULO:

De todo lo dicho se deduce la conveniencia de emplear una metodologia que permita incrementar el rendimiento de los diseñadores y la fabilidad del sistema final. Esta metodologia aplicada al diseño de SBMP debe incluir los siguientes aspectos:

- Descomposicion del sistema global en subsistemas mas sencillos, definiendo claramente las iteraciones entre ellos a traves de los argumentos transferidos, tal como se hace en el HIPO.

- Descripcion de los algoritmos en sucesivos niveles, gradualmente mas detallados, evitando incluir demasiadas estructuras de control en una sola descripcion, tal como recomienda el metodo de JACKSON.

- Adoptar en lenguaje secuencial para la descripcion del algoritmo antes de comenzar la escritura del programa. De esta forma se tendra una descripcion del algoritmo a nivel de lenguaje de programacion, pero independiente de este. Ademas, puesto que el objetivo de esta descripcion es documentar el programa final, se pueden evitar los problemas de interpretacion traduciendo las palabras clave del lenguaje al castellano. Las estructuras de control de lenguaje deberan adaptarse a las del PASCAL, pues ya veremos que ello facilita la puesta a punto del sistema.

3.- HERRAMIENTAS PARA LA PUESTA A PUNTO DE SISTEMAS.-

INTRODUCCION.-

Como ya se ha dicho anteriormente, la puesta a punto de programas es la unica que ha suscitado polemicas y que ha llegado a conclusiones razonadas. Por esta razon, y dado que la puesta a punto del sistema supondra el 50% del tiempo total de construccion del prototipo, se ha considerado conveniente la inclusion, al principio del capitulo, de un resumen de las tecnicas de puesta a punto mas generalizadas. Ello nos permitira fijar las caracteristicas de la metodologia relativas a la puesta a punto de los circuitos y los programas.

Tambien se ha comentado en la introduccion de esta parte de la obra, las herramientas para la puesta a punto de SBMP son las unicas que han experimentado un avance desde la implantacion generalizada de los microprocesadores. La creciente complejidad de las posibilidades ofrecidas por estos nuevos utiles, hace imprescindible la adopcion de nuevas tecnicas de puesta a punto que aprovechen al maximo estas posibilidades. Por ello se han analizado en este capitulo las caracteristicas de las herramientas de puesta a punto: Programas monitores y depuradores, analizadores logicos y emuladores de circuitos. De este analisis se deduciran las caracteristicas que debe tener la metodologia de puesta a punto adoptada para el diseño de SBMP, asi como las

ventajas e inconvenientes presentados por cada una de las herramientas disponibles en el mercado, para su aplicacion al campo de los microprocesadores.

3.1.- OBJETIVOS GENERALES DE LA PUESTA A PUNTO.-

El objetivo basico de la puesta a punto es, evidentemente, demostrar que el programa funciona, pero esta afirmacion es un tanto imprecisa, pues se puede demostrar que el programa funciona en ciertas condiciones.pero demostrar que funciona en todas es materialmente imposible. Esta consideracion permite r varios procedimientos de modificacion. En uno de ellos se emplea el distinguir varios procedimientos de verificacion. En alguno de ellos se emplea el ordenador,como herramienta de soporte, y vamos a analizar las ventajas que se pueden obtener de su uso.

3.1.1.- PROCEDIMIENTOS PARA LA VERIFICACION DE PROGRAMAS.-

Dependiendo de las garantias que se quieran obtener de la validez de un programa, se pueden usar cuatro procedimientos:

3.1.1.1.- VERIFICACION EXHAUSTIVA.-

Requiere la ejecucion del programa siguiendo cada uno de los caminos posibles al menos una vez. Tiene el claro inconveniente de que usa muchas horas de ordenador, especialmente en programas con bucles.

3.1.1.2.- VERIFICACION SELECTIVA.-

Se ejecuta el programa con un juego de datos seleccionado por el programador, el cual se supone que recoge todos los casos significativos. Sin embargo es dificil demostrar que todos los casos singulares han sido verificados, por lo cual no se puede garantizar la correccion del producto.

3.1.1.3.- COMPROBACION FORMAL.-

Consiste en realizar calculos inductivos sobre el algoritmo, para obtener las aserciones, los predicados y los rangos de variacion de las variables en cada tramo del programa . Esta aproximacion no es viable en grandes sistemas, pues requiere una buena base matematica y no permite considerar casos especiales como el procesado paralelo.

3.1.1.4.- SISTEMAS AUTOMATICOS DE EVALUACION DE PROGRAMAS.-

Son los que utilizan el ordenador para llevar a cabo

alguno de los procedimientos anteriores de verificacion. Se ha demostrado que permiten mejorar el tiempo de UCP y el de desarrollo en un 25-30 %.

3.1.2.- HERRAMIENTAS AUTOMATICAS PARA LA VERIFICACION.-

Existe una gran variedad de herramientas para la verificacion de programas. Las diferencias entre unas y otras son muy variadas, por ello se pueden emplear varios criterios de clasificacion, pero tal vez mas interesante sea el basado en la funcion realizada por la herramienta.

3.1.2.1.- ANALISIS ESTATICO DEL PROGRAMA FUENTE.-

No requiere ejecucion del programa sujeto del analisis. Es realizado por el programa ensamblador, compilador, o interprete, segun los casos. Puede abarcar los siguientes aspectos:

* Analisis sintactico del codigo.

* Verificacion de la estructura del programa; generacion del grafo ; verificacion de la correccion de las estructuras; verificacion del final de los bucles.

* Verificacion de los interfaces entre los modulos: numero y tipo de las variables de entrada y salida.

* Verificacion de la secuencia de sucesos: Sentencias inalcanzables; etiquetas no referenciadas; uso de variables no definidas previamente; sentencias sin sucesores; anidado incorrecto de bucles...

3.1.2.2.- ANALISIS DINAMICO DEL PROGRAMA FUENTE.-

El programa se ejecuta para comprobar su comportamiento y hacer estadisticas. Se deben insertar monitores, los cuales afectan a la ocupacion de memoria, el tiempo de ejecucion, y en ocasiones a las estructuras de control.

Puede comprender los siguientes puntos:

* Monitorizacion del comportamiento del programa durante la ejecucion: evolucion del estado de los registros; evolucion de las variables ubicadas en memoria; evolucion de las entradas/salidas ...

* Generacion automatica de casos de comprobacion.

* Verificacion de las aserciones. Comprende: Verificacion de rangos de variables y subindices de matrices; Verificacion de entradas y relaciones entre variables; Verificacion de la verosimilitud de los datos de entrada; verificacion inversa del resultado, comprobando su adecuacion con los datos.

* Insercion de defensas del programa.

3.1.2.3.- MANTENIMIENTO DEL PROGRAMA.-

Se entiende por mantenimiento de un programa, no solo las operaciones a realizar sobre el, para conseguir "repararlo" cuando se ha "estropeado", sino tambien el conjunto de documentacion necesaria para modificarlo. Asi resultan ser dos los aspectos a tener en cuenta en la puesta a punto de un programa, desde el punto de vista del mantenimiento:

* Generacion de la documentacion del programa: Tabla de referencias cruzada de variables y sentencias; tabla de secuencia de llamadas a subrutinas; Tabla de referencia cruzada de bloques comunes y subrutinas; grafo del programa.

* Validacion de las modificaciones. Para ello se pueden emplear cualquiera de las tecnicas descritas.

3.1.2.4.- MEJORA DE LAS PRESTACIONES.-

Aunque en general no se pueda considerar propiamente como parte de la puesta a punto; en algunos casos puede ser imprescindible, para conseguir que el programa se ajuste a especificaciones de velocidad u ocupacion de memoria.

Se puede conseguir de dos formas:

* Reestructurando el programa: Eliminando comprobaciones redundantes, variables innecesarias, saltos...

* Extrayendo y validando operaciones en paralelo.

CARACTERISTICAS		DEPURADOR	ANAL LOGICO	EMULADOR
Visualizar	memoria	si	si	si
Modificar	registros	si	si	si
Puntos ruptura		si	si	si
Desensamblado		opcional	opcional	si
Traza	paso a paso	si	si	si
	selectiva	no	si	si
	tiempo real	si	si	si
Redireccionado menor		no	no	si

Tab. 3.1.- Caracteristicas de las herramientas de puesta a punto de sistemas basados en microprocesador.

3.2.- PROGRAMAS DEPURADORES.-

Antes de comenzar a describir los programas depuradores, tal vez deberiamos hablar de los ensambladores y compiladores existentes, pero sus carateristicas no difieren de las descritas en forma general, al hablar de los procedimientos de analisis estatico.

Los programas depuradores permiten un analisis dinamico de los programas del usuario. En general constan de una serie de rutinas de utilidad que son disparadas por el programador cuando se interrumpe la ejecucion del programa. La mayoria de depuradores proporcionan las siguientes prestaciones:

3.2.1.- VISUALIZACION DE LA MEMORIA.-

Permite verificar la evolucion de las variables almacenadas en la memoria. Los llamados "depuradores simbolicos" permiten referenciar las posiciones de memoria mediante la etiqueta o el nombre que se le ha dado en el ensamblador o lenguaje de alto nivel. Con ello se evita tener que calcular las direcciones efectivas cada vez que se monta el programa de nuevo, despues de haber introducido alguna modificacion.

La informacion contenida en la memoria, se presenta habitualmente en hexadecimal, pero tambien se puede

obtener decodificada en ASCII o desensamblada, para comprobar el correcto funcionamiento de la UCP o la sincronizacion con una puerta de E/S.

3.2.2.- VISUALIZACION DE LOS REGISTROS INTERNOS.-

Esta prestacion es algo mas compleja que la anterior, pues implica que los registros deben ser transferidos a memoria y de alli presentados al operador. Basicamente tiene la misma utilidad que la prestacion anterior, pues permite comprobar la correcta evolucion de las variables "temporales" de las rutinas del programa.

El registro contador de programas, en particular, permite el seguimiento de la "traza" del programa, es decir la verificacion de la rama del programa, que se esta ejecutando.

3.2.3.- PUNTOS DE RUPTURA DE PROGRAMA.-

Permiten detener la ejecucion del programa del usuario antes de llegar al final logico, devolviendo el control al programa depurador o al monitor del sistema. Brindan al operador la posibilidad de ejecutar el programa por tramos, comprobando al final o al principio de la ejecucion de cada tramo si se cumplen las relaciones previstas entre las variables.

3.2.4.- DESENSAMBLADO Y ENSAMBLADO DE INSTRUCCIONES EN LINEA.-

Ya hemos comentado la posibilidad de visualizar el contenido de las posiciones de memoria desensamblado. Algunos depuradores simbolicos permiten, ademas, el ensamblado de instrucciones sueltas, para sustituir a otras instrucciones del programa, sin necesidad de reensamblarlo todo. El sistema tiene la limitacion de no permitir insertar mas instrucciones de las ya existentes, pues al no ser el codigo reubicable, no se puede desplazar el programa en la memoria.

3.2.5.- EJECUCION PASO A PASO.-

Permite ejecutar las instrucciones del programa de una en una. Normalmente despues de la ejecucion de cada instruccion se visualiza el estado de los registros, con lo que se va obteniendo la traza del programa. Tiene el inconveniente de que la ejecucion no es tiempo real.

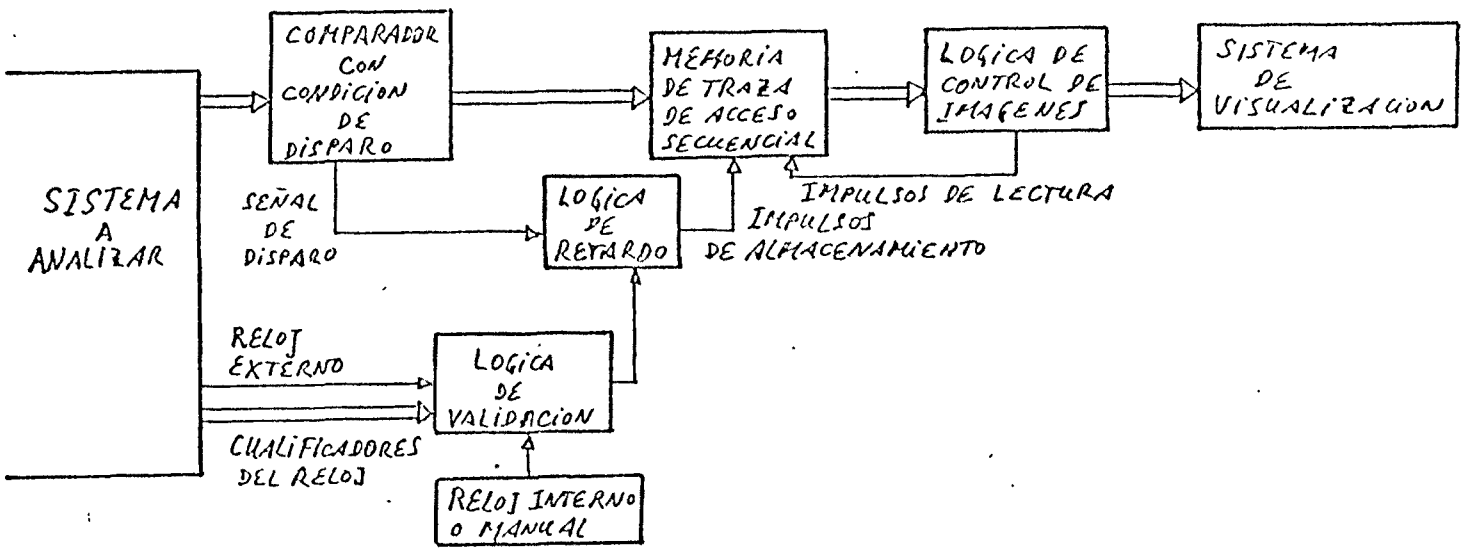


Fig. 3.1.- Diagrama de bloques de un Analizador Logico. La memoria de la Traza permite estudiar detenidamente los datos recogidos entorno a un instante determinado, aunque no sean repetitivos. Las logicas de validacion del reloj y retardo permiten definir el instante entorno al cual se almacenaran los datos y las condiciones que se deben dar para almacenar uno de ellos.

3.3.- ANALIZADORES LOGICOS.-

Los analizadores logicos comenzaron a desarrollarse a principios de los anos 70, cuando los microprocesadores habian comenzado a ser empleados por un gran numero de empresas, que necesitaban herramientas mas modernas para la puesta a punto de los circuitos logicos.

Basicamente un analizador logico es un aparato de medida que recoge muestras de una serie de señales logicas en paralelo, almacenandolas en forma de 1 o 0 segun sea el valor de la entrada en el instante de muestreo. Hasta aqui no se aprecian mejoras respecto a un osciloscopio de memoria, pues la diferencia radica en la forma de disparo, la presentacion de los valores almacenados y el numero de canales, que oscila entre 8 y 48.

3.3.1.- PALABRA DE DISPARO.-

Tal vez sea esta la innovacion mas destacada de los analizadores logicos. Consiste en referir el proceso de recogida y presentacion de muestras, al instante en que todas las señales muestreadas toman un valor preseleccionado, llamado palabra de disparo. Esta caracteristica contrasta con la condicion de disparo de un osciloscopio, en la que tan solo influye uno de los canales de entrada.

Para cada bit de la palabra de disparo se pueden especificar tres estados o condiciones: bajo, alto o indeterminado. En el ultimo caso de bit no influye en el calculo de la condicion de disparo.

Algunos analizadores logicos tienen palabras de disparo con menor numero de bits que el total de canales de entrada. Otros analizadores permiten definir mas de una condicion de disparo, iniciandose el proceso de recogida de datos cuando se cumple una cualquiera de las dos.

En los analizadores mas sencillos, se puede hacer coincidir el instante en que se cumpla la palabra de disparo con el primer dato almacenado, con el ultimo o con el central. En los mas complejos, se puede especificar un retardo variable entre el instante de disparo y el primer o ultimo dato almacenado, con lo cual se pueden analizar procesos en los que el lapso de tiempo entre la detencion del error y su origen es muy grande comparado con la capacidad de almacenamiento del analizador.

3.3.2.- CUALIFICACION DE LOS DATOS.-

Entendemos por cualificacion de los datos, el conjunto de condiciones que se deben cumplir para que se almacene un dato en la memoria del analizador.

Basicamente el analizador se dedica a tomar muestras

del estado de las lineas de entrada de datos y almacenarlas en una memoria tampon. Por tanto hara falta una señal que actue de reloj, uno de cuyos flancos se usara para fijar el instante en que se almacenen los datos. Este reloj puede ser proporcionado por el analizador, en cuyo caso sera periodico, o bien ser externo al analizador. En este ultimo caso, la señal empleada como reloj no tiene por que ser periodica, puede ser una señal de validacion de datos o direcciones del bus del procesador o la propia señal del reloj del sistema analizado.

Cuando el proceso a analizar abarca mas impulsos de reloj que posiciones tiene la memoria tampon del analizador, se tienen dos opciones para almacenar informacion de todo el proceso. Si es repetitivo se pueden ir introduciendo retardos respecto a la condicion de disparo. Si no es repetitivo o si el reloj tiene una frecuencia mayor que la de los datos significativos del proceso, interesa especificar una condicion de cualificacion del reloj. Esta condicion de cualificacion suele ser una operacion booleana entre dos señales cualificadoras. De esta forma, cuando se produzca un flanco activo del reloj, solo se almacenaran los datos en memoria si se cumple la condicion de cualificacion, ello permite separar direcciones y datos en los buses multiplexados, o almacenar tan solo las transferencias a un banco de datos de la memoria, ignorando los codigos de instruccion que se extraen de la memoria de programa entre dos accesos al banco de datos.

3.3.3.- PRESENTACION DE LOS DATOS.-

Una de las caracteristicas mas apreciadas de los analizadores logicos es la variedad de formas de presentacion de los datos, campo en el que se han realizado los avances mas destacados. Se pueden distinguir tres modos de presentacion de los datos:

3.3.3.1.- NUMERICA.-

Como su propio nombre indica, en esta forma de presentacion los datos almacenados en la memoria tampon, se muestran codificados en un sistema de numeracion. Los sistemas mas empleados son, logicamente, el hexadecimal, el octal y el binario, en este orden.

Normalmente, junto a los datos almacenados se indica la informacion recogida en el instante en que se cumplio la palabra de disparo. Ello proporciona un punto de referencia en la coleccion de datos presentados. En algunos casos se destaca la parte de la informacion recogida en la memoria tampon que difiere de la almacenada en una memoria patron. Esta posibilidad es especialmente apreciada en los puestos de control de calidad o en montajes en cadena, pues permite ver rapidamente si existe alguna divergencia con un sistema patron previamente memorizado.

3.3.3.2.- ALFANUMERICA.-

La version mas sencilla se limita a dar la informacion decodificada segun el codigo ASCII, cuando esto es posible, al estilo de los depuradores de programas. Tan solo es util en los sistemas de comunicacion con el hombre.

Los analizadores mas inteligentes y pensados especialmente para trabajar con sistemas basados en un microprocesador determinado, permiten separar los codigos de instruccion de los datos y presentan los primeros dsesensamblados, en forma de nemotecnicos, y los datos en forma numerica. Con esto se consigue que la presentacion de los datos recogidos tenga el mismo aspecto que el listado del programa que se esta ejecutando en el microprocesador.

3.3.3.3.- TEMPORAL.-

En este epigrafe se incluyen todos los modos de presentacion basados en un eje de tiempos: cronograma. De esta forma la presentacion es analoga a la que ofrece un osciloscopio de muestreo.

Entorno a esta idea basica de presentacion de la informaion, existen variantes que aumentan las prestaciones del sistema:

* Posibilidad de cambiar el numero de muestras presentadas simultaneamente en la pantalla. Equivalentea cambiar la fase de tiempos en un osciloscopio.

* Visualizacion de impulsos mas cortos que el periodo de la señal de muestreo. Esta caracteristica es especialmente util en la depuracion de prototipos, pues permite detectar los impulsos espureos consecuencia de los distintos retardos de las señales respecto al reloj maestro.

* Identificacion de niveles indefinidos. Esta prestacion indica de forma especifica las muestras que tienen un nivel superior al cero logico e inferior al uno logico. Ello pone de manifiesto posibles causas de errores, como consecuencia de sobrecarga de señales o desacoplo en los buses.

3.3.3.4.- COMBINACIONAL.-

Consiste en presentar la informacion muestreada por el analizador logico en forma grafica, sobre un sistema de coordenadas X,Y. Para ello se toman los valores binarios de las entradas para calcular las coordenadas del punto de la pantalla que se debe iluminar.

Este sistema de visualizacion de la informacion es especialmente util cuando se desea analizar el funcionamiento de un procesador del que se desconoce el

comportamiento. Para ello basta con calcular las coordenadas del punto a visualizar a partir de las señales del bus de direcciones del procesador, lo cual permite saber en todo momento cuales son las direcciones de memoria a las que se esta accediendo. De esta forma se puede deducir cual es el fragmento de programa que se esta ejecutando o el tamaño de la pila del procesador ò del area de datos con la que se esta trabajando.

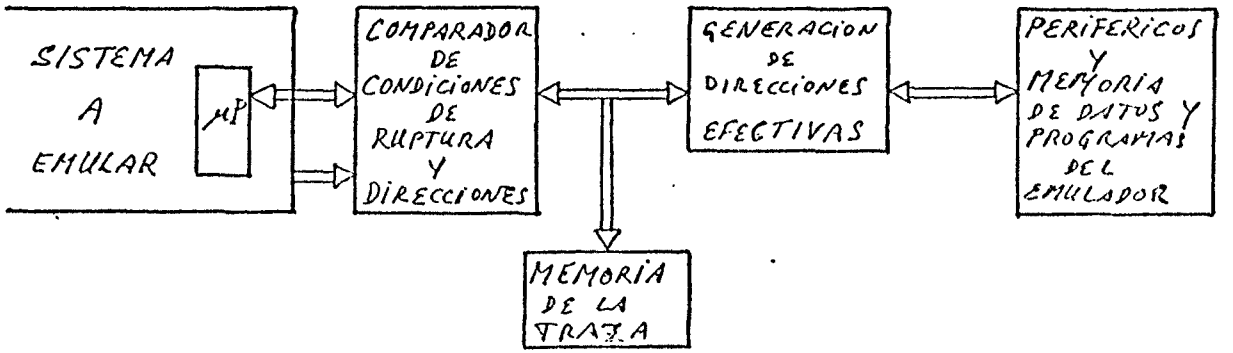


Fig. 3.2.- Diagrama de bloques de un Emulador de Circuito (EDC). Las condiciones de ruptura permiten detener al procesador para verificar el contenido de los registros internos. Los circuitos comparador y generador de direcciones permiten acceder a posiciones de memoria y periféricos distintas de las generadas por el procesador, y encaminarlas hacia el sistema a emular o hacia el EDC.

3.4.- EMULADORES DE CIRCUITO (EDC).-

Los emuladores de circuito (EDC) reunen las prestaciones de los depuradores y de los analizadores logicos en un solo equipo. Permiten analizar el estado del sistema estatica y dinamicamente, es decir con el procesador "parado" o ejecutando el programa del usuario, tal como se haria con un depurador o un analizador logico, respectivamente. En definitiva son unos sistemas capaces de supervisar y controlar la evolucion de un microcomputador.

Se presentan en dos tipos de versiones. Las mas sencillas son portatiles y permiten el analisis de los sistemas acabados, simplemente sustituyendo la pastilla del microprocesador por el pulpo del EDC. Las mas sofisticadas forman parte de un sistema de desarrollo de microprocesadores (SDM) y cuentan con posibilidades nuevas, respecto a las de los analizadores y depuradores, que los hacen mas utiles en el proceso de conjecucion de circuitos y programas del sistema a depurar.

3.4.1.- MAPEADO DE MEMORIA Y E/S.-

Esta es la caracteristica mas destacada de los EDC frente a la simple union de un analizador y un depurador. Consiste en controlar el bus de direcciones del procesador analizado, de forma que se puede acceder a la memoria del sistema o del emulador, segun la conveniencia

del operador. El multiplexado puede ser fijo durante toda la ejecuciom del programa del usuario, o depender del banco de memoria o la puerta referenciados.

Esta prestacion permite ejecutar sobre la RAM del EDC los programas que finalmente deberan estar ubicados en la ROM del sistema emulado, con el consiguiente ahorro de tiempo en la comprobacion de cada modificacion. En algunos casos el multiplexado incluye un desplazamiento de las direcciones, lo que permite ejecutar en la zona intermedia de la memoria del EDC rutinas de interrupciones, y en general programas que deberian ubicarse en la ROM del EDC.

3.4.2.- DEPURACION DE LOS CIRCUITOS DEL INTERFAZ.-

En los sistemas de desarollo de Microcomputadores SDM convencionales, la depuracion de los programas no puede tener en cuenta los perifericos reales del sistema final, salvo que este sea el propio SDM. Ello significa que, forzosamente, se deben simular las E/S sobre las del SDM, con el inconveniente de que no se pueden hacer ejecuciones en tiempo real, ni tener en cuenta las condiciones de contorno especificas del sistema, con los consabidos problemas de ruidos y asincronismos imprevistos.

El EDC permite ejecutar los programas utilizando todos o parte de los circuitos de interfaz del sistema a depurar, con independencia de la ubicacion de los

programas en el SDM o el sistema del usuario (SU). Ello tiene varias ventajas:

* Se pueden comenzar a probar los circuitos de interfaz antes de tener todo el SU montado, lo cual permite paralelismos en el trabajo.

* Se pueden ejecutar programas de verificacion sobre la RAM del EDC, con el consiguiente ahorro de reprogramaciones de las memorias EROM del S.U. Esta ventaja tiene, no obstante, el inconveniente de no permitir la ejecucion en tiempo real, pues el uso de una memoria o E/S distinta de la del S.U. implica ciclos de acceso a memoria y perifericos mas largos.

* En la comprobacion de sistemas en fase de produccion o en su mantenimiento "en el campo", se pueden ejecutar programas de verificacion de los interfaces del SU sobre el EDC, los cuales permiten comprobar aisladamente cada circuito del S.U.

3.4.3.- ANALIZADOR LOGICO EN TIEMPO REAL.-

Esta es una opcion que incluyen casi todos los EDC en mayor o menor medida. Basicamente consiste en almacenar las transferencias que se realizan a traves de las patillas del procesador emulado. A estos analizadores logicos incluidos en los EDC, se pueden aplicar las mismas consideraciones hechas sobre los analizadores

logicos en general. No obstante conviene resaltar algunas peculiaridades, consecuencia del hecho de estar especializados en un microprocesador concreto:

* Las palabras de disparo se especifican como operaciones del procesador, a nivel de instruccion, como los puntos de ruptura de los depuradores, o a nivel de ciclo de maquina, especificando simplemente el tipo de transferencia (Lectura o escritura en memoria o periferico), la direccion y el dato.

* Las palabras de disparo pueden constituir tambien puntos de ruptura, ampliandose de esta forma las posibilidades de definicion de estos.

* La ejecucion se puede analizar paso a paso, o en tiempo real. En este ultimo caso los datos almacenados en la memoria tampon, "traza", se observan despues de detener la ejecucion mediante uno de los puntos de ruptura, palabras de disparo, o manualmente.

* La traza se puede presentar tal como se ha almacenado en cada ciclo de reloj o de la maquina, o bien en forma desensamblada, agrupando todos los ciclos de ejecucion de una misma instruccion.

* Si conviene se pueden cualificar las transferencias a almacenar, para ahorrar memoria tampon si el proceso a analizar es muy largo. Para la cualificacion se puede emplear cualquiera de los datos recogidos: control,

direccion o datos del bus.

* En algunos casos, ademas se pueden recoger datos externos de cualquier punto del interfaz que se este analizando. Ello permite comprobar al grado de sincronismo entre el programa que se esta ejecutando y los estimulos externos. Esta posibilidad es especialmente util en los sistemas multiprocesadores, pues permite obsersevar las transiciones de los semaforos de sincronismo o los indicadores de los mensajes entre procesadores.

El analizador logico, como componente de un EDC, es la herramienta ideal para la puesta a punto del conjunto formado por los programas y los circuitos del sistema del usuario.

RESUMEN:

De todo lo dicho se deduce la conveniencia del empleo de analizadores logicos o emuladores de circuito en la puesta a punto de SBMP, pues son las unicas herramientas que permiten realizar un seguimiento exhaustivo del comportamiento del sistema, sin interferir en su comportamiento autonomo. Interesa resaltar ademas la posibilidad que ofrecen de "congelar" un fragmento determinado de la ejecucion del algoritmo, para poderlo analizar detenidamente. El aprovechamiento de esta posibilidad requiere definir las caracteristicas, que distinguen al instante inicial y final del tramo que deseamos analizar, para definirselo concretamente al analizador o al emulador. Esta caracterizacion de determinados puntos del algoritmo sera uno de los objetivos primordiales de la parte puesta a punto de la metodologia propuesta.

CONCLUSIONES DE LA PARTE I

La comparacion de las herramientas para el diseño de circuitos y las de programacion, permite apreciar un gran desequilibrio entre ambas, pues los diseñadores de circuitos siguen basando los diseños en la imaginacion del diseñador, sin apenas asistirle en la concepcion global del circuito. Los programadores, en cambio, cuentan con unas metodologias de diseño que permiten ir simplificando los problemas en pasos sucesivos hasta llegar a niveles de complejidad accesibles a cualquier tecnico. Ademas, la extrema complejidad de los programas producidos ha forzado a los analistas de programas a planificar la puesta a punto de los sistemas, intentando de esta forma garantizar al maximo la fiabilidad del producto final.

Todo ello nos lleva a la conclusion de que, si se debe adoptar una metodologia comun al diseño de las partes cableada y programada del SBMP, esta metodologia debe partir de las tecnicas empleadas en el diseño de programas, adaptadas a las caracteristicas de los SBMP, y adaptandolas tambien al diseño de los circuitos especifico del sistema, sistematizando al maximo esta fase del diseño, pra hacerla igualmente accesible a tecnicos no excesivamente experimentados.

P A R T E II

METODO BARNA PARA EL DISEÑO DE SISTEMAS LOGICOS BASADOS EN MICROCOMPUTADOR

Los objetivos del metodo, tal como se ha indicado en la introduccion, se pueden resumir en cuatro puntos fundamentales:

* Actualizacion de la pedagogia del diseño de sistemas logicos, adaptandola a las necesidades impuestas por los crecientes niveles de integracion de las pastillas.

* Actualizacion de las metodologias de diseño de sistemas logicos, adaptandolas para el aprovechamiento maximo de los sistemas programables.

* Documentacion uniforme de las partes programada y cableada de un sistema, haciendo patentes las interrelaciones entre ambas partes.

* Aprovechamiento de las herramientas de depuracion de sistemas para el analisis simultaneo de las partes programada y cableada.

Para conseguirlo, se han intentado unificar al maximo los metodos empleados en los mundos de los circuitos y los programas, inconexos hasta la aparicion de los microprocesadores. Esta unificacion implica un analisis descendente, modular y jerarquizado de todo el sistema, descomponiendolo en tareas, que finalmente se programaran

o cablearan, intentando que esta decision se tome lo mas tarde posible, y que no sea irreversible. Para ello la sintesis de las tareas se hace tambien descendente, pasando sucesivamente por descripciones en forma de:

* Lenguaje descriptor de algoritmos

* Lenguaje de alto nivel

* Lenguaje de transferencia de registros

* Lenguaje descriptor de circuitos

Evidentemente, tan solo la descripcion de las tareas cableadas llegara hasta el ultimo nivel de detalle, las tareas programadas se quedaran en uno de los dos anteriores, dependiendo del lenguaje empleado en la programacion.

En el capitulo 4 se hace una descripcion general del metodo, haciendo hincapie en los tres primeros objetivos descritos. En el capitulo 5 se analizaran las ventajas del metodo descrito para la consecucion del ultimo objetivo.

En la descripcion del metodo no se ha hecho excesivo enfasis en la definicion de los lenguajes empleados en los distintos niveles de descripcion, pues no se ha considerado que fueran una parte importante del metodo. Ademas la recomendacion de un determinado lenguaje abriria una polemica entorno a su idoneidad, fundamentalmente

entre quienes en la actualidad esten habituados a otros lenguajes, restando de esta forma importancia a la concepcion global del metodo, la cual es adaptable a cualquiera de los lenguajes de descripcion de programas o circuitos utilizados actualmente.

4.- CONCEPCION DEL SISTEMA.-

En este capitulo se afronta la primera parte del diseño, correspondiente a la concepcion de la realizacion final. Esta concepcion comprende dos fases bien diferenciadas: Analisis y Sintesis.

La primera tiene como objetivo final la generacion de un algoritmo que satisfaga los requerimientos del problema planteado. La segunda fase es la sintesis del sistema que ejecute el algoritmo generado; el objetivo final de esta fase es el conjunto de esquemas logicos y listados de programas de la maquina que resolvera el problema inicialmente planteado.

Ambas fases se desarrollaran sucesivamente, pero abarcando simultaneamente todo el sistema. De esta forma se evitara un desdoblamiento prematuro del sistema en una parte cableada y otra programada. Al mismo tiempo se tiene la ventaja de obtener un unico tipo de descripcion para todas las tareas que debe realizar el sistema lo cual permitira aplicar tambien una sola metodologia a la puesta a punto de todas las tareas.

La fase del analisis comienza con una descomposicion del problema global en otros mas sencillos. Esta descomposicion se llevara a cabo en pasos sucesivos, siguiendo las tecnicas ya descritas de JacKson e HIPO. A estos fragmentos o parcelas del sistema global es alo que

denominaremos "tareas", no en el sentido estricto en el que se emplea este termino en la programacion concurrente, sino en el sentido de distinguir los distintos trabajos o acciones mas o menos independientes que debera ejecutar el sistema, ya sea mediante un programa, ya mediante unos circuitos especificos. Finalmente se describiran cada una de las tareas de forma grafica y (o) secuencial. Estas descripciones se realizaran siguiendo un camino descendente y por aproximaciones sucesivas, evitando de esta forma dar grandes saltos, que hagan dificil la asimilacion de la descripcion.

La fase de sintesis abarca los distintos procesos de traduccion de estas descripciones de las tareas a los lenguajes de programacion o esquemas logicos que constituiran la realizacion final del sistema. Estas traducciones se realizaran a uno u otro lenguaje en funcion de la clarificacion realizada en el momento del analisis. Como se vera, una de las ventajas del metodo es precisamente la sencillez y automatismo con que estas traducciones se pueden llevar a cabo, lo cual simplifica posibles retoques en la asignacion de procedimiento de realizacion optima para algunas tareas conflictivas, ya sea por ocupacion de memoria, de procesador o por problemas de sincronismo con procesos externos.

4.1.- ANALISIS DESCENDENTE.-

En este apartado nos limitaremos exclusivamente al

problema de la descomposicion del sistema en tareas, dejando para apartados sucesivos el resto de los aspectos del analisis del sistema. Esta descomposicion se hara siguiendo dos criterios, el primero modular y el segundo funcional. La descomposicion funcional se llevara a cabo por aproximaciones sucesivas, siguiendo las tecnicas de diseño descendente mencionadas en el capitulo 2.

4.1.1.- DESCOMPOSICION DEL PROBLEMA.-

El analisis del problema implica una descomposicion del mismo en subproblemas mas sencillos y abordables. De esta forma se evita que la complejidad del problema inicial impida considerar cada uno de los pequeños detalles que comprende. La descomposicion del problema implica un desdoblamiento en tareas, que se puede realizar siguiendo dos criterios fundamentales:

4.1.1.1.- DESCOMPOSICION FUNCIONAL.-

Consistente en patentizar o aislar cada una de las funciones que deben realizar la maquina final en general, o las tareas o funciones en que se va descomponiendo. Esta sucesiva fragmentacion de los fragmentos, conduce finalmente a una lista de tareas realizadas por el sistema operativo, por programas de la libreria, o por los interfaces o perifericos a incluir en la maquina final. Todos ellos ligados entre si por una estructura que se desprende directamente de las sucesivas descomposiciones y que constituye el cuerpo del algoritmo a sintetizar.

4.1.1.2.- DESCOMPOSICION MODULAR.-

Consistente en considerar la maquina final compuesta por una serie de modulos con funciones especificas, mas o menos independientes, cada uno de los cuales contribuye

en mayor o menor grado a la resolucion del problema. Dentro de cada modulo se definen una serie de tareas afines, y el problema global se describe en terminos de esas tareas que componen los distintos modulos de la maquina. Asi, por ejemplo, para analizar un problema de gestion de ficheros, tendremos unos modulos de entrada/salida por pantalla, salida por impresora, acceso a los discos o cintas, etc...; y el algoritmo final consistira en una estructura en cuyas ramas habra llamadas a las tareas de los modulos indicados, como: escribir caracter en pantalla, escribir linea en pantalla, escribir texto en pantalla, leer caracter, etc.

4.1.2.- CONCEPTO DE ANALISIS DESCENDENTE.-

Este concepto surge a partir de la descomposicion funcional. Si estudiamos detalladamente este metodo, nos podemos dar cuenta de que comenzamos por expresar el problema general en terminos de tareas mas concretas. Estas, a su vez, se detallan en terminos de otras subtareas aun mas concretas, y asi hasta llegar a descripciones a nivel de transferencia de bit.

Dicho de otra forma, si suponemos la maquina final compuesta por una sucesion de maquinas de distintos niveles, anidadas una dentro de otra, cuya lista, de exterior a interior, puede ser (fig.4.1):

1.- Unidad de control de proceso, interfaces de entrada/salida.

2.- Microprograma.

3.- Rutinas manejadoras de los interfaces.

4.- Rutinas de utilidad del sistema operativo.

5.- Rutinas de utilidad de la aplicacion.

6.- Programas de la aplicacion.

7.- Aplicacion.

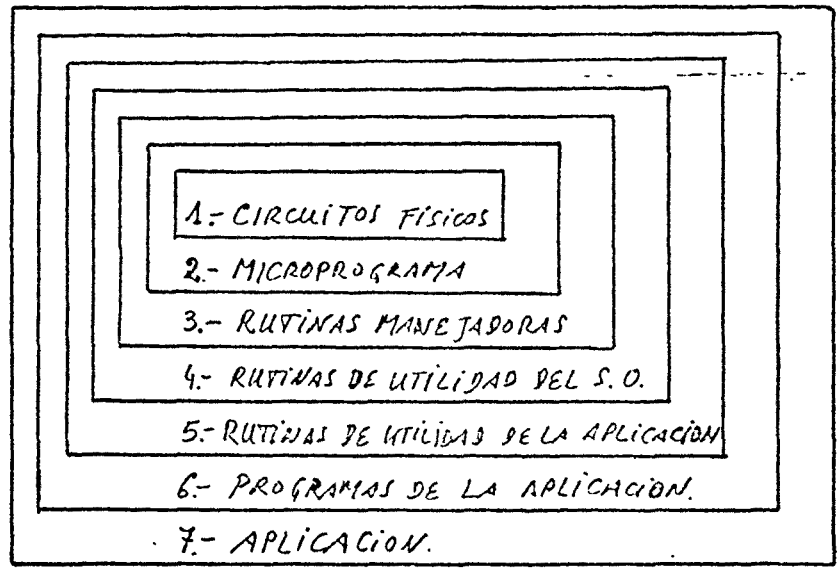


Fig. 4.1.- Anidamiento de niveles de comprensión de las tareas en que se descompone un sistema basado en microprocesador (SPIT). Las tareas de los niveles más externos utilizan o controlan las de los más internos.

Cada una de estas maquinas se caracteriza por las operaciones que es capaz de efectuar , y las ordenes que, a su vez, emite a las maquinas de nivel inmediatamente inferior. Ello implica una relacion superior/inferior entre las operaciones efectuadas por una u otra maquina, de la que hablaremos mas adelante. Esta relacion es la que nos permite calificar de descendente al analisis, pues va describiendo cada tarea u operacion de una maquina en funcion de operaciones o tareas de maquinas inferiores. De esta forma se van obteniendo descripciones cada vez mas detalladas o profundas de la aplicacion que se desea realizar.

El analisis descendente se apoya en dos elementos fundamentales:

* El analisis de la estructura de los datos que se van a manejar, como primer criterio de descomposicion del problema.

* La libreria de programas de utilidad del sistema fisico que vamos a emplear en la realizacion de la maquina; pues dentro de lo posible procuraremos descomponer las tareas en otras contenidas en la libreria, puesto que las descripciones de niveles inferiores de estas son conocidas y por tanto no es necesario profundizar mas en el analisis de dichas tareas.

Evidentemente la libreria estara formada tanto por las rutinas de utilidad proporcionadas por el fabricante

como por aquellas que nosotros hayamos definido para aplicaciones anteriores o especificamente para esta realizacion.

Tal como se ha dicho en un capitulo anterior, el uso de una metodologia basada en estos dos principios supone un ahorro del 50% en los costes del desarrollo de los programas.

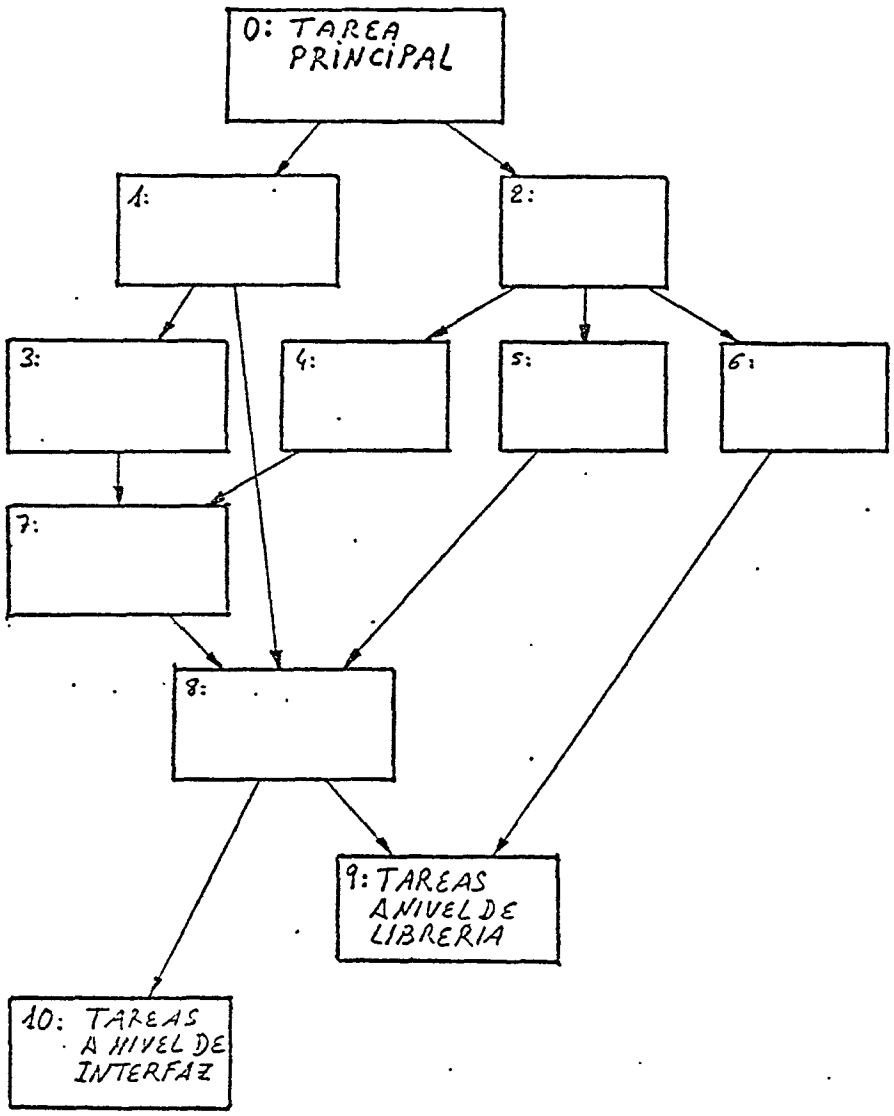


Fig. 4.2.- Organigrama jerarquico de las tareas en que ha sido descompuesto el sistema. Las flechas van de tarea utilizadora a utilizada.

4.2.- JERARQUIZACION DE LAS TAREAS.-

Una vez establecidos los criterios de descomposicion del sistema, vamos a resaltar las relaciones existentes entre las tareas en que ha quedado descompuesto. Ello nos permitira agrupar las tareas horizontal y verticalmente, esto es modular y funcionalmente, de forma que se minimicen las interrelaciones entre los grupos formados.

4.2.1.- ORGANIGRAMA JERARQUICO.-

En el apartado anterior hemos visto que se podia establecer una relacion de dependencia entre las tareas del sistema. Esta relacion nos permite establecer una ordenacion de las tareas, desde la mas importante, la global que las abarca todas y que estara en el vertice superior del organigrama; hasta la tarea encargada de conseguir que se ponga a 1 o a 0 una determinada linea de salida, que no se puede descomponer en tareas mas sencillas y que estara en la base del organigrama, dando soporte a todas las tareas que la usan.

Al conjunto de tareas ordenado segun la relacion de utilizacion existente entre ellas, es a lo que llamaremos organigrama jerarquico. (fig.4.2.)

Para construir el organigrama basta con ir colocando las tareas en sucesivas filas, comenzando por la mas general en la fila superior y finalizando con las mas

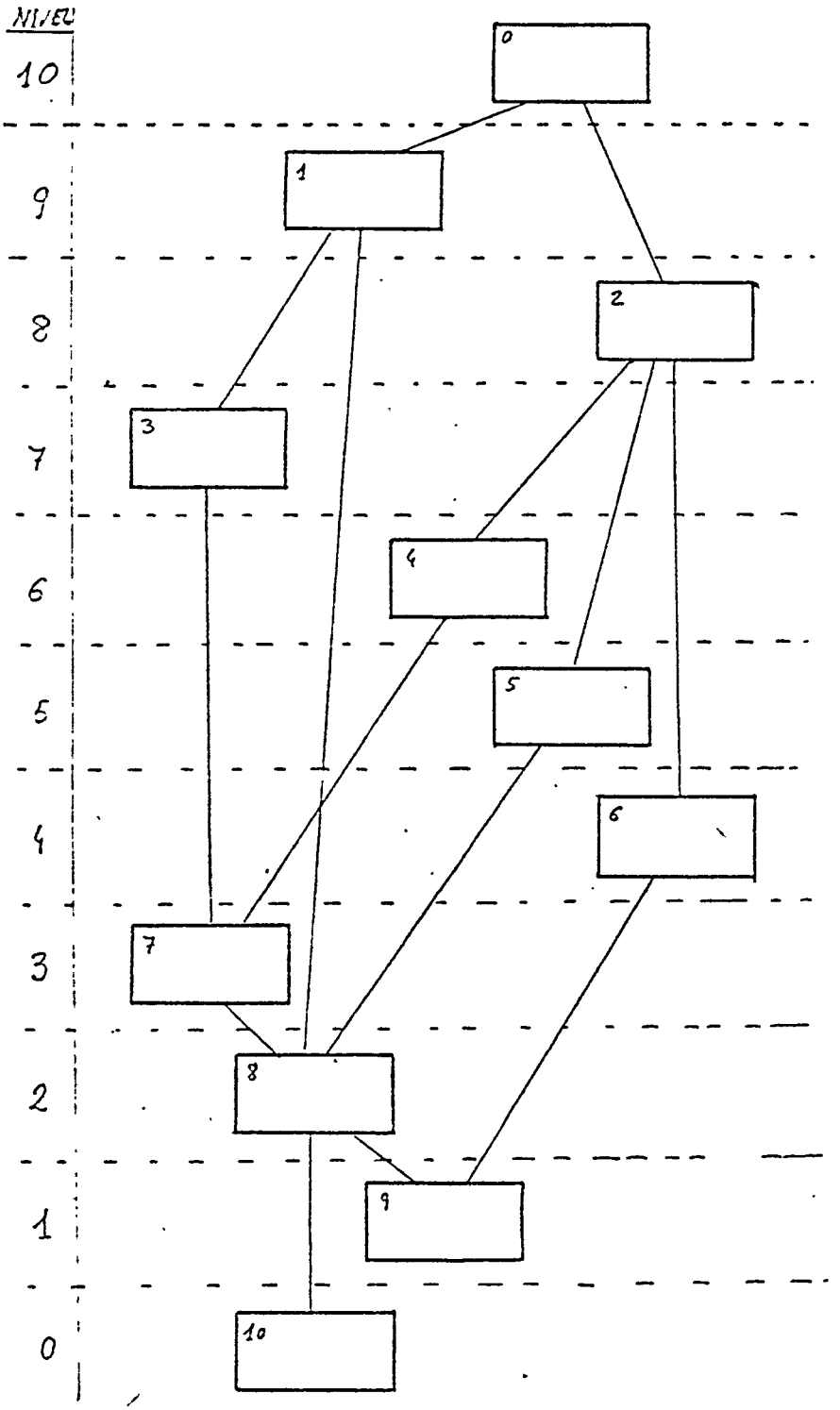


Fig. 4.3.- Organigrama jerarquico mostrando el maximo numero de niveles, que coincide con el numero de tareas.

concretas en la ultima fila.

4.2.2.- NIVELES JERARQUICOS.-

Llamamos nivel jerarquico a cada una de las filas en las que se han ido colocando las tareas en que se ha descompuesto el sistema.

El numero de niveles de un sistema depende, evidentemente, del numero de tareas que colocamos en cada uno de ellos. El maximo coincide con el numero de tareas, pues no tiene sentido un nivel sin tarea (fig.4.3). El numero minimo de niveles coincide con la longitud del "camino critico", entendiendo por tal aquel que une la tarea mas general con una de las mas particulares, pasando por el mayor numero de tareas intermedias (fig.4.4). Ello es debido a que hemos partido del supuesto de que existe una relacion de subordinacion entre las tareas y no se permite colocar una tarea subordinada en un nivel igual o superior al de otra de orden superior.

Esta restriccion se puede cumplir en la mayoria de los casos, salvo cuando se planifican corrutinas que se llaman entre si. En este caso colocaremos las corrutinas en un mismo nivel, considerandolas como una sola tarea a efectos de calculo del camino critico.

Dado que una tarea puede ser utilizada desde otras muchas, entre dos tareas puede haber dos o mas caminos de

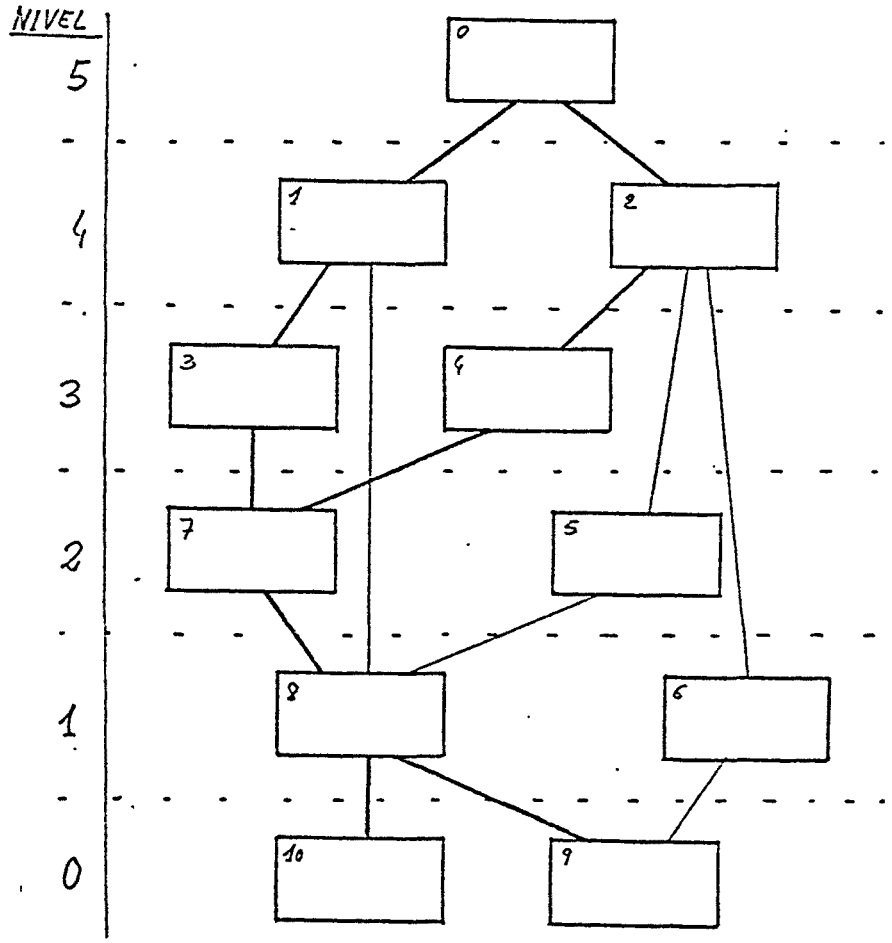


Fig. 4.4.- Organigrama jerarquico mostrando las tareas agrupadas en el minimo numero de niveles posible. Este numero coincide con el de tareas que comprende el camino critico, o mas corto posible, que une la tarea principal con la mas alejada de ella.

longitudes distintas, entendiendose por longitud de un camino el numero de tareas por las que pasa. Esto significa que los caminos mas cortos uniran tareas de niveles no adyacentes, por lo cual estas tareas podran ser desplazadas del nivel que ocupan a alguno de los intermedios, siempre que no se violen las relaciones de orden establecidas con el resto de las tareas. Asi en el ejemplo de la fig.4.4 las tareas 5 y 6 se podran elevar hasta el nivel 3 pero en cambio la tarea 2 no puede descender al nivel 3, porque ello violaria su relacion con la tarea 4, que a su vez no puede bajar de nivel, por estar en un camino critico.

4.2.3.- CRITERIOS PARA LA AGRUPACION EN NIVELES.-

Puesto que tanto el numero de niveles, como la eleccion del nivel que debe ocupar una tarea, no estan todavia fijados, debemos establecer unos criterios que nos permitan su determinacion.

Al hablar de analisis descendente, se ha dicho que todo sistema se puede considerar compuesto por una serie de maquinas anidadas, de tal forma que las mas externas se basan en las mas internas. Uno de los objetivos de la agrupacion en niveles de las tareas en las que previamente ha sido descompuesto el sistema, es hacer patentes las maquinas en las que se basa, de forma que cada nivel constituira una maquina con un nivel de abstraccion determinado y distinto del de los niveles adyacentes.

Asi pues, dentro del criterio global de agrupar en un mismo nivel tareas con un grado de abstraccion similar, podemos enumerar algunos criterios mas objetivos y por tanto mas faciles de aplicar en la practica.

4.2.3.1.- COMPLEJIDAD DE LAS ESTRUCTURAS DE DATOS.-

En una primera aproximacion, agruparemos las tareas en funcion de las estructuras de datos que manejen, puesto que los datos impondran su estructura a los programas que los traten. De hecho a cada estructura de datos le corresponderan una serie de tareas, ligadas entre si y con las correspondientes a las otras estructuras de datos mediante relaciones similares a las existentes entre estas estructuras. (fig.4.5).

4.2.3.2.- PROXIMIDAD A LOS PERIFERICOS.

Este criterio nos permite hacer una agrupacion en niveles mas coherentes con el grado de abstraccion absoluto de las tareas. Asi se evita que se agrupen en los niveles mas bajos del organigrama jerarquico, tareas con objetivos muy dispares , tales como sincronizacion con perifericos y llamadas a rutinas de una libreria, cuyo grado de abstraccion es bien distinto. Analogamente deberan ocupar niveles distintos, tareas de inicializacion de pastillas de control y tareas de manipulacion de datos de estrada/salida conectadas directamente a un periferico,

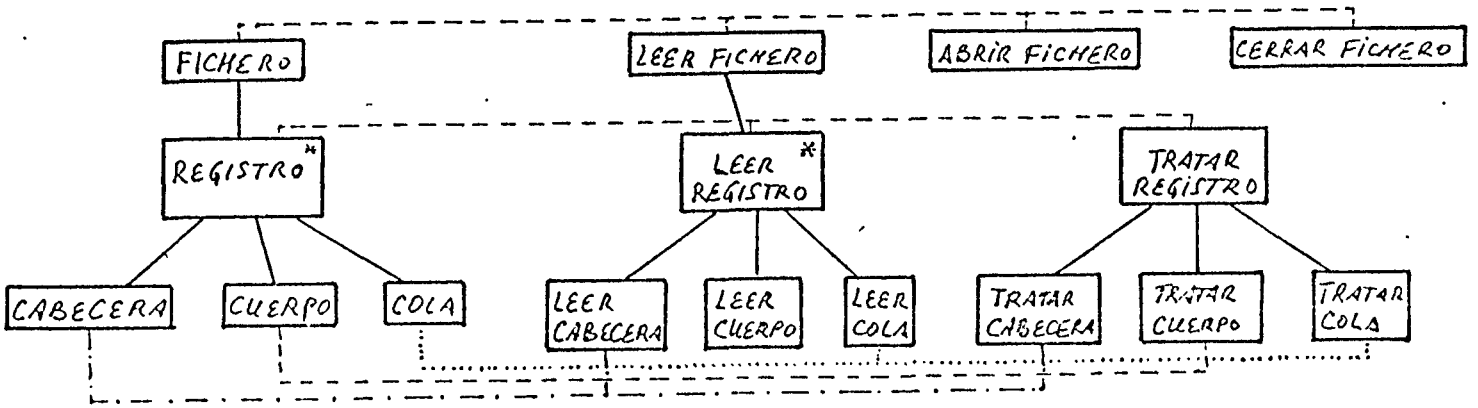


Fig. 4.5.- Relaciones entre las estructuras de datos y de tareas.

pues maneja circuitos con un nivel de complejidad distinto.

4.2.3.3.- COMPLEJIDAD DE LAS ESTRUCTURAS DE CONTROL.-

Este criterio se fundamenta en el supuesto de que la complejidad de la estructura de control refleja directamente la complejidad global de la tarea.

La forma en que finalmente sera realizada la tarea nos impone un determinado grado de abstraccion de la descripcion. Debemos por tanto aplicar criterios de agrupacion que nos indiquen "a priori" el procedimiento optimo de realizacion de la tarea: Lenguaje de alto nivel (LAN), ensamblado, (LBN), circuitos MSI o circuitos SSI. Si bien el procedimiento empleado finalmente en la realizacion estara fijado por criterios economicos o tecnologicos, uno de los objetivos de la agrupacion en niveles es indicar unas prioridades de realizacion por uno u otro metodo. Asi las tareas mas generales seran las candidatas prioritarias para ser realizadas en LAN, en tanto que las mas concretas lo seran a una realizacion mediante circuitos SSI, y las de los niveles intermedios deberian ser realizadas en ensamblador (LBN) o con circuitos MSI.

Es evidente que los procedimientos mas potentes se aplicaran a las tareas mas dificiles de realizar y la dificultad de realizacion de una tarea esta directamente

relacionada con la complejidad de su estructura de control, tanto si se debe programar como si se debe cablear. Ello justifica la aplicacion de este criterio para la ordenacion de las tareas en niveles y ademas nos va a permitir asociar la barrera programa/circuito a una de las divisiones entre niveles, pues habra ciertas garantias, "a priori", de que las tareas sobrepuestas a esa division son mas dificiles de controlar que las inferiores.

4.2.3.4.- FRECUENCIA DE UTILIZACION.-

Este criterio no tiene absolutamente nada que ver con el nivel de abstraccion de la tarea, y su objetivo es precisamente optimizar la utilizacion del procesador y de los circuitos asociados a tareas especificas.

Asi las tareas ejecutadas esporadicamente subiran de nivel, para franquear la barrera LAN/LBN. En cambio las tareas que consumen mayor tiempo de procesador bajaran de nivel, franqueando las barreras LAN/Ensam. y (o) programa/circuito (P/C).

De esta forma el LBN nos permitira optimizar el tiempo de ejecucion de las tareas que mas pesan en el computo del tiempo total consumido. La realizacion de las tareas mas frecuentes mediante circuitos hace que los circuitos esten activos la mayor parte del tiempo, y puesto que durante ese tiempo el procesador puede dedicarse a otras tareas, ello da mayor rendimiento al sistema en conjunto.

En este ultimo caso aparece el problema de sincronizacion de tareas que se ejecutan en paralelo, una programada y otra(s) cableada(s). Este problema complica el trabajo de diseño en proporcion al grado de cooperacion de las tareas, y aqui solo lo analizaremos someramente mas adelante.

4.2.4.- DETERMINACION DE LA BARRERA PROGRAMA/CIRCUITO.-

Como ya se ha apuntado anteriormente, la colocacion de esta barrera es crucial en el rendimiento del trabajo de diseño. Su determinacion es funcion de la eleccion de un compromiso entre el precio de los circuitos y la velocidad de respuesta del sistema. Esta relacion precio/velocidad en general aumenta con la velocidad, como ejemplo baste citar la posibilidad que existe de aumentar la rapidez del sistema simplemente utilizando memorias o procesadores de ciclo mas corto, sin modificar la barrera P/C.

Puesto que los procesadores y memorias mas rapidos implican un coste del sistema sensiblemente mayor, ademas de tener una limitacion concreta para el procesador elegido, en muchas ocasiones nos veremos obligados a ganar tiempo realizando un mayor numero de tareas por circuito, lo que equivale a trasladar la barrera P/C a un nivel superior, con el consiguiente aumento en el coste del sistema, al tener que añadirle circuitos especializados en la realizacion de tareas especificas.

En el caso contrario de que el procesador este sobrado de tiempo, se pueden realizar mas tareas por programa, desplazando la barrera P/C hacia abajo, con lo que se ahorran circuitos especializados y se consigue un sistema mas compacto.

4.2.4.1.- SITUACIONES LIMITE.-

Dado que practicamente todas las tareas se podran realizar por programa o circuito, la situacion ideal de la barrera no esta fijada "a priori". En principio tan solo estan determinados los limites superior e inferior que puede ocupar la barrera programa/circuito.

4.2.4.1.1.- LIMITE INFERIOR.-

Corresponde al caso en que se realizan por programa el mayor numero de tareas posible. En este caso se realizaran por circuito exclusivamente las tareas que se puedan ejecutar sin circuiteria de control, o bien controladas por una pastilla MSI especializada. Salvo en este ultimo caso todo el control se realizara por programa y los circuitos especificos del sistema se reduciran a adaptaciones de las pastillas MSI y los registros de E/S a los buses del procesador.

4.2.4.1.2.- LIMITE SUPERIOR.-

Constituye al extremo opuesto al anterior y no esta tan definido como este, pues si consideramos como posible el empleo de microprocesadores monopastilla, convencionales o a rebanadas para realizar circuitos 'microprogramados" especializados en la realizacion de algunas tareas, en el limite estas tareas pueden llegar a ser todas y el sistema convertirse en un ordenador subordinado del que ejecuta el sistema operativo o el monitor. Este caso es bastante frecuente entre los sistemas de desarrollo universales, en los que coexisten varios subsistemas independientes, que en ocasiones actuan en paralelo:

* Procesado de palabras: Monitor, editor, base de datos

* Depuracion de programas: Compilador, ensamblador, depurador.

* Depuracion de sistemas: Emulador.

No obstante la situacion normal del limite superior de la barrera P/C sera aquella en la que coexistan un procesador principal, al que estaran asignadas las tareas de planificacion mas generales, y unos procesadores subordinados o maquinas microprogramadas, especializados en la ejecucion de tareas especificas tales como calculo matricial, artimetica de coma flotante, E/S, etc...

4.2.4.2.- PROCEDIMIENTOS DE DETERMINACION DE LA SITUACION OPTIMA DE LA BARRERA P/C.-

Como se ha dicho en el apartado anterior, la situacion optima de la barrera no se podra conocer hasta que no se hayan descrito todas las tareas a nivel de LTR, o incluso hasta que no se hayan realizado los programas y circuitos del prototipo. No obstante el hecho de que hasta el momento solo se ha detallado la forma de obtener una descripcion de las tareas a nivel de bloques jerarquizados, vamos a describir ahora toda la parte del proceso de diseño que afecta a la determinacion de la barrera P/C, extrayendola del resto, del proceso, que se describira mas adelante.

4.2.4.2.1.- SUPUESTO DE PARTIDA.-

Inicialmente asumiremos el limite inferior de la barrera. Ello tiene la ventaja de permitirnos diseñar un primer prototipo con el minimo de esfuerzo, pues el procedimiento de realizacion de una tarea por programa es mas corto que el de sintesis cableada. Realmente podemos considerar dos limites inferiores, segun tengamos en cuenta o no las limitaciones temporales del entorno. Caso de no tenerlas en cuenta, las tareas de mas bajo nivel tal vez no se puedan comprobar con las condiciones de contorno reales, pero al menos se podra verificar la correccion del algoritmo asociado a la tarea, simulandolo en el procesador principal. Caso de poderse controlar por

programa el medio externo, tan solo se deberan convertir a circuito las tareas que colapsen el tiempo del procesador, o las que nos permitan un ahorro considerable de memoria, pero como ya veremos estos cambios no supondran excesivo esfuerzo de diseño adicional; ademas se realizaran una vez verificados los algoritmos por programa, con lo cual los posibles errores de la tarea cableada se limitaran a problemas de cableado o adaptacion entre bloques funcionales.

4.2.4.2.2.- CONSIDERACIONES FISICAS REALES.-

Con el supuesto de partida se detallan las descripciones de las tareas hasta llegar a nivel de programa o de LTR, segun esten por encima o no de la barrera P/C, respectivamente. Este nivel de detalle de la descripcion de las tareas nos permite determinar la ocupacion de memoria y la capacidad de reaccion de los programas a los estimulos externos, asi como la viabilidad de los circuitos a cablear. En las tareas programadas se debera verificar:

* Que no transcurra demasiado tiempo entre una accion exterior y la reaccion deseada del sistema.

* Que se puedan contar intervalos de tiempo con la suficiente precision.

* Que se puedan generar respuestas o estimulos con

suficiente rapidez y precision en la temporizacion.

En las tareas cableadas se tendran que hacer las mismas comprobaciones para determinar la calidad de componentes a emplear. Ademas, la complejidad de las estructuras de control puede aconsejar el diseño de una maquina microprogramada, o el empleo de un propcesador auxiliar mas rapido que el principal. En este caso aparece una segunda barrera, que podriamos denominar microprograma/circuito, que subdivide las tareas consideradas cableadas desde la optica del sistema global.

De estas consideraciones se deducira la consolidacion de la barrera inicial o su desplazamiento hacia arriba, englobando mas tareas en la parte programada. El desplazamiento inverso no es normal pues hemos partido del limite inferior. En cualquier caso, el desplazamiento de la barrera se hace progresivamente, alternandolo con las comprobaciones indicadas anteriormente, para determinar las ventajas obtenidas con el trasvase de cada nueva tarea, facilitando de esta forma posibles vueltas atras de algunas tareas, una vez rebasado el objetivo fijado.

4.2.4.2.3.- AJUSTE FINAL.-

El objetivo final de este proceso de aproximaciones sucesivas, para la determinacion de la situacion idonea de la barrera P/C, es obtener un sistema que cumpliendo

con las especificaciones del entorno, tenga un coste minimo. Este coste minimo implica un maximo aprovechamiento de la memoria ROM, llenando los huecos con las tareas que no tienen una temporizacion critica, aunque ello suponga simplemente el ahorro de un biestable. Tambien se puede dar el caso contrario, en el cual el paso de una tarea de programa a circuito permite ahorrar una pastilla de memoria, a cambio tal vez de un contador o un biestable.

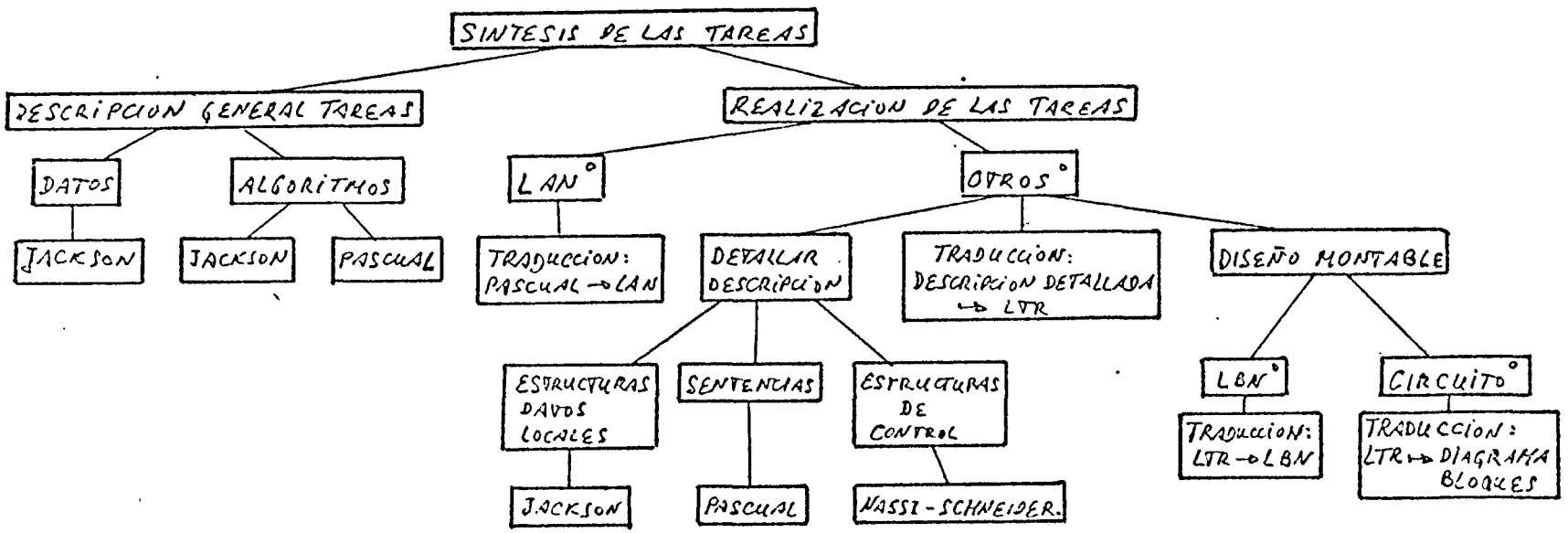


Fig. 4.6.- Diagrama de la secuencia de sintesis. La descripcion general de datos y algoritmos sirve de base a la traduccion a lenguaje de alto nivel (LAN) o a lenguaje de transferencia de registros (LTR). Esta ultima, a su vez, constituye la base del resultado final de la realizacion, ya sea un programa en lenguaje de bajo nivel (LBN), ya un circuito logico.

4.3.- SINTESIS DE LAS TAREAS.-

Una vez se han ordenado las tareas, en las que se ha descompuesto el sistema, procederemos a su sintesis siguiendo un metodo descendente. En este capitulo enumeramos los pasos que se deben seguir, explicando la metodologia a aplicar en cada uno de ellos. En la fig. 4.6. se muestra un diagrama de la secuencia de sintesis propuesta.

4.3.1.- DESCRIPCION DE LAS ESTRUCTURAS DE LOS DATOS.-

Como paso previo a la descripcion algoritmica de cada una de las tareas, conviene hacer una descripcion de las estructuras de datos que se van a manejar. Existen dos razones para ello, la primera es que las propias estructuras de datos constituyen una primera aproximacion a la estructura del algoritmo que debe ejecutar la tarea, tal como demuestra Jackson.

La segunda razon es que antes de describir el algoritmo de cada tarea, debemos concretar los mecanismos de comunicacion entre las mismas, especificando las informaciones que interpretara y generara cada tarea; y puesto que estas informaciones se deberan almacenar en variables comunes o en argumentos de transferencia, hay que tener definidos estos para poder concretar las partes de los algoritmos dedicadas a las comunicaciones con otras tareas.

En realidad las informaciones a transmitir entre las tareas se deducen de las relaciones existentes entre ellas desde el mismo momento de la concepcion de cada tarea. En esta fase de la sintesis se concretan:

* Los tipos de datos a transferir.

* Las variables que serviran de soporte para esas transferencias.

* Los valores limite de los datos a transferir.

La especificacion de estos valores limite sera especialmente util en caso de contar con un lenguaje que permita definir tipos del usuario, como el PASCAL, y en el momento de calcular las aserciones correspondientes a los puntos de entrada y salida de las tareas, a fin de verificar su correcto funcionamiento.

Para la descripcion de las estructuras de los datos se emplearan los diagramas propuestos por Jackson (2.3). En estos diagramas los datos se describiran hasta el nivel de detalle apropiado al metodo de realizacion previsto para la tarea. No obstante, puesto que los diagramas de Jackson fuerzan una descripcion en niveles de detalle crecientes, los distintos niveles de descripcion conseguidos para los datos segun la tarea no limitan posteriores replanteos en el metodo de realizacion. Caso de descender el nivel de realizacion de la tarea bastara con seguir concretando la descripcion de

los datos a partir del punto en que se interrumpio. Caso de ascender el nivel de realizacion bastara con no considerar la parte mas detallada de la descripcion de los datos, sino conviene.

4.3.2.- DESCRIPCION DE LOS ALGORITMOS.-

La descripcion de los algoritmos mediante los cuales se realizaran las funciones encomendadas a las tareas se llevara a cabo en dos fases.

4.3.2.1.- DIAGRAMAS DE JACKSON.-

La primera fase consistira en una descripcion general del algoritmo en forma de diagrama de Jackson. Esta primera descripcion tiene la ventaja de forzar a un analisis descendente de cada tarea, proporcionando al mismo tiempo una documentacion grafica bien estructurada y de rapida interpretacion. Cuando las estructuras de datos que se deben manejar sean muy complejas, estos diagramas ayudan a conseguir una primera descomposicion del algoritmo, pues esta se correspondera con alguna o algunas de las descomposiciones realizadas sobre los datos al describir sus estructuras, como consecuencia de las cuales ya se han obtenido diagramas de Jackson correspondientes a los datos.

4.3.2.2.- PASCAL-NATURAL.-

El objetivo de esta segunda fase es obtener una descripcion secuencial del algoritmo ya descrito graficamente segun Jackson. Esta nueva descripcion servira de puente entre la descripcion grafica y el lenguaje de alto nivel empleado en su programacion, o las descripciones mas detalladas que seguiran caso de no ser un LAN el metodo de realizacion escogido para la tarea.

Esta descripcion secuencial del algoritmo se hara en lo que se ha dado en llamar Pascal-natural, y que se define a continuacion. Dado que este lenguaje debe ser facilmente comprensible estructurado y facil de utilizar, se ha pensado en adaptar alguno de los lenguajes descriptivos de algoritmos existentes, eliminando todas las trabas que para su uso suponen las imposiciones sintacticas del compilador. El lenguaje resultante de estas consideraciones tiene las estructuras de control del PASCAL con las palabras clave en la lengua vernacula del usuario: castellano, catalan, gallego, etc. Ello unido a una absoluta flexibilidad en la denominacion de los procedimientos y definicion de los tipos y rangos de las variables, permite emplear esta descripcion como comentario de los diagramas de Jackson, constituyendo el conjunto una inmejorable documentacion de las tareas a sintetizar.

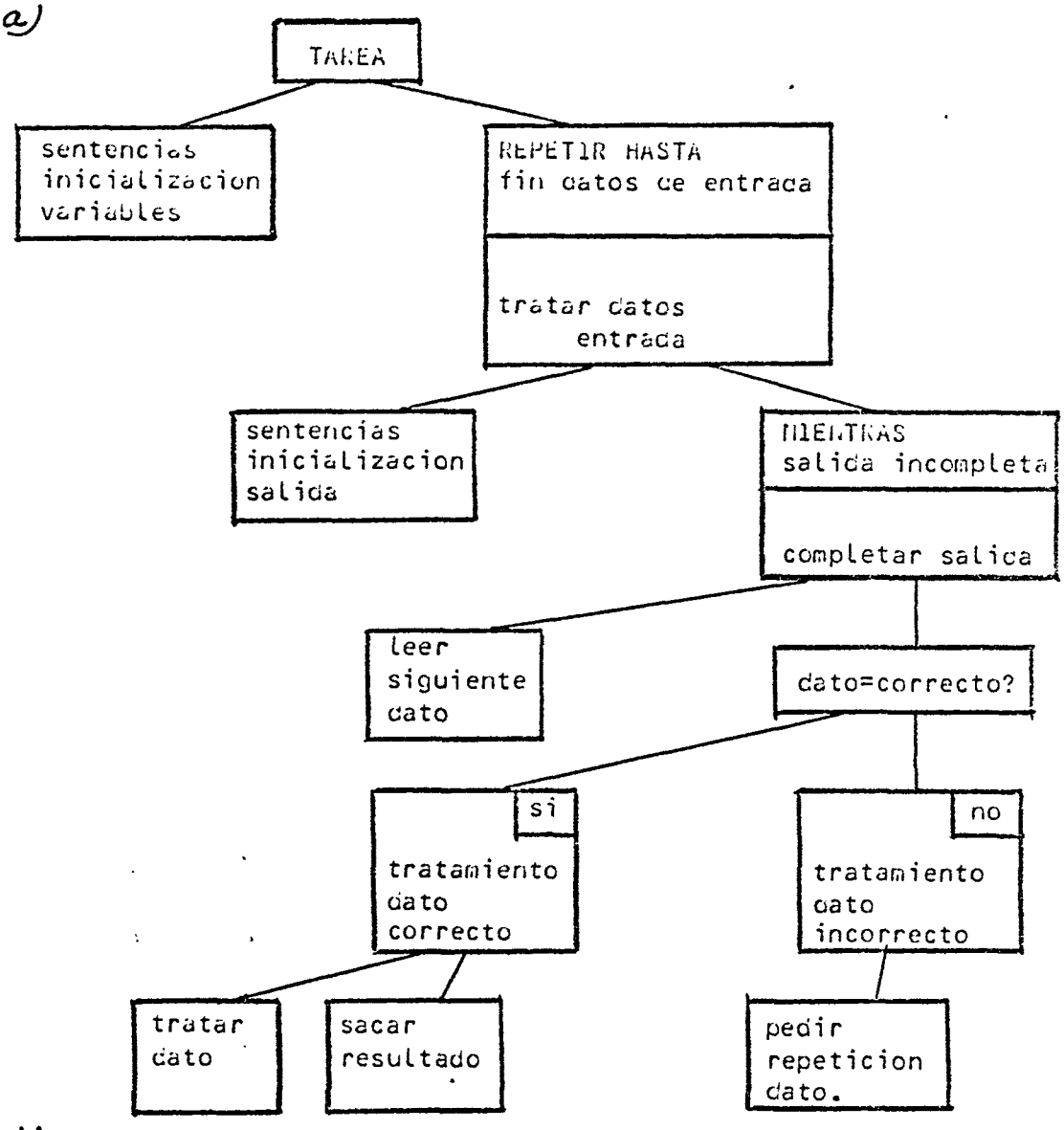
La eleccion de las estructuras del PASCAL y no las del PDL o del ADA radica en la extensa bibliografia que

basa sus ejemplos en el Pascal, y a la gran cantidad de microprocesadores que cuenten con compilador PASCAL, lo cual en muchos casos reducira el trabajo de programacion a una simple traduccion literal, añadiendo la informacion requerida por el compilador y que resultaria obvia a un lector humano. En las fig. 4.7.a. y 4.7.b se pueden observar las equivalencias entyre las descripciones de un algoritmo segun Jackson y en Pascal-natural.

El grado de flexibilidad y simplicidad del Pascal-natural empleado dependen de la practica del diseñador. En el ejemplo de la fig.4.7.b se han suprimido al maximo las palabras llave COMIENZO y FIN, sustituyendolas por un indentado que deja bien patente la estructura de control a la que pertenece cada sentencia. Tambien se han suprimido los "punto y coma" que separan las sentencias en PASCAL, pues no hay dificultad en diferenciarlas con una simple lectura, a pesar de emplear varias palabras para denominar una sola tarea (tratar el dato, sacar el resultado,...).

El objetivo del lenguaje empleado es permitir una descripcion "literaria" lo mas rigurosa.clara y sencilla posible. Una opcion seria traducir directamente al lenguaje vernaculo del diseñador las palabras clave del Pascal, u otro lenguaje de descripcion de algoritmos, lo cual tendria la ventaja de ser absolutamente inambiguo, a cambio del inconveniente de ser redundante para un lector humano. A esta segunda opcion de lenguaje descriptor de algoritmos le llamaremos PASCUAL, permitiendonos una traduccion libre del castellano del nombre del lenguaje

a)



b)

```
0) Tarea (arg. entrada, arg. salida)
1)     sentencias de inicializacion de variables
1)     REPETIR
2)         sentencias inicializacion de la salida
2)         MIENTRAS la salida este incompleta HACER
3)             leer el siguiente dato
3)             Si el dato leido es correcto
4_5)                 ENTONCES tratar el dato
  5)                     sacar el resultado
4_5)                 SINO pedir repeticion del dato
1)     HASTA fin de los datos de entrada
0) Fin de tarea.
```

Fig. 4.7.-
Descripcion general de una tarea:
a) Segun el metodo de Jackson.
b) Segun el Pascal-natural. Obsérvese la correspondencia entre el nivel de detalle de la descripcion en Jackson y el nivel de indentacion de la descripcion en Pascal-natural.

de programacion del que se han tomado las estructuras de control.

El lenguaje PASCAL-natural se describe en el Apendice I. Se caracteriza por tener las mismas estructuras de control que el Pascal, difiriendo de este en que las palabras clave abren un nuevo nivel de descripcion mas detallado, o lo cierran. Las sentencias pertenecientes a un determinado nivel se caracterizan por estar escritas a partir de la columna de tabulacion correspondiente a ese nivel, de esta forma las palabras clave sirven de puente entre niveles de tabulacion, dejando claro el tipo de estructura a que corresponde cada nivel. En la fig. 4.7.b se puede observar el resultado de traducir a este lenguaje el algoritmo descrito en forma de diagrama de Jackson en la fig.4.7.a. En ambas figuras se han numerado los niveles de descripcion, pudiendose observar el completo paralelismo existente en los dos casos.

En las tareas mas triviales se puede suprimir una de las dos descripciones, si se considera que la documentacion queda suficientemente clara con una sola de ellas.

En ambas formas de descripcion, la claridad de la misma esta condicionada a que se pueda presentar en una sola pagina. Esto que en principio se podria considerar como una limitacion, es en realidad una ventaja, pues fuerza a definir tareas suficientemente sencillas como

para que sean facilmente comprensibles para todos los colaboradores en el proyecto, y faciles de modificar en el futuro, de darse el caso.

```
leer_dat
escr_dat
trat_dat
Tarea (arg_entr; integer, arg_sal:character)
blanc CONSTANT:' '; ret CONSTANT: 13;
VAR: dato, long INTEGER;
BEGIN
    arg_sal:=blanc
    REPEAT
        long:=1; arg_sal:= arg_sal+CHR(ret)
        WHILE long <= 70
        DO BEGIN leer_dat(dato)
               IF dato >=0
                 THEN IF dato <=9
                        THEN BEGIN
                                 trat_dat; long:=long+1;
                                 arg_sal:=arg_sal+dato;
                                 arg_entr:=arg_entr-1
                             END
                        ELSE escr_dat(CHR(21))
                 ELSE escr_dat(CHR(21))
            END
    UNTIL arg_ent=0
END
```

Fig. 4.8.- Codificacion en PASCAL de la tarea descrita en la fig. 4.7. Esta codificacion se obtiene realizando simplemente una traduccion literal de aquella descripcion.

```
10  REM TAREA
20  REM arg_entrada=A0 , arg_sal=A$
30  REM dato= D$ ; long= L
40  REM leer_dato= 1000
50  REM escribir_dato= 1100
60  REM tratamiento_dato= 1200
70  A$ = " "
80  REM tratar datos entrada
90  L=1
100 A$= A$+ CHR$(13)
110 REM completar salida
120 IF L> 70 THEN 230
130 GOSUB 100
140 IF D$ <'0' THEN 200
150 IF D$ >'9' THEN 200
160 GOSUB 1200
170 A$= A$+D$
180 A0= A0-1
190 GOTO 220
200 D$ = CHR$(21)
210 GOSUB 1100
220 GOTO 120
230 IF A0<>0  THEN 90
240 RETURN
```

Fig. 4.9.- Codificacion de BASIC de la tarea descrita en la fig. 4.7. Esta codificacion requiere una adaptacion de las estructuras de control del algoritmo, dado que el BASIC no es un lenguaje de los llamados estructurados.

4.3.3.- TRADUCCION A LAN.-

Las tareas descritas en Pascal-natural que deben ser realizadas en lenguaje de alto nivel seran las unicas traducidas al correspondiente lenguaje de programacion, terminando asi sus fases de diseño. El resto de tareas seguiran el resto del proceso a partir de la descripcion a Pascal-natural.

En esta fase de traduccion a LAN vamos a distinguir dos casos, segun sean las sentencias de control admitidas por el lenguaje. En las figs. 4.8. y 4.9. se muestran dos posibles traducciones del algoritmo descrito en la fig. 4.7., empleando dos LAN con distintas primitivas de control.

4.3.3.1.-LENGUAJES CON PRIMITIVAS DE CONTROL COMPLEJAS.-

Dentro de este calificativo un tanto ambiguo, se incluiran los lenguajes que cuentan con primitivas de control de bucle flexibles:

* Repetir/hasta

* Mientras/hacer

* Hacer/repetir/salir/fin.

Y con sentencias condicionales de numero indefinido

de opciones:

* Caso/de

Ademas, evidentemente, de las primitivas de control tipicas de los lenguajes de alto nivel:

* Iteracion con contador (para/hasta/siguiente).

* Condicionales de 2 o 3 opciones.

Lenguajes de este tipo son por ejemplo: PASCAL,PLM, PLZ, MPL, ADA,...

En este caso la tarea de traduccion se limita a una adaptacion sintactica de la descripcion. Esta adaptacion implicara basicamente tres fases:

4.3.3.1.1.- CORRECCION DE LAS PALABRAS CLAVE E INCLUSION DE LOS SEPARADORES DE SENTENCIAS O BLOQUES.-

Esto supone una simple traduccion literal de la informacion contenida en la descripcion en Pascal-natural. Solamente en el caso de querer optimizar el codigo generado se podran sustituir llamadas a subtareas por el codigo de estas, cuando solo sean utilizadas desde ese punto, y conservando el concepto de la tarea a nivel de comentario en el programa fuerte.

Los comentarios del programa fuerte deberan cumplir una doble mision. Primeramente destacaran claramente los comienzos y finales de los bloques ejecutables, que aparecen en cada uno de los niveles de descripcion de la tarea segun Jackson, o en su defecto segun el Pascal-natural. Ello permitira deshacer facilmente el camino desde cualquier punto del programa, en la fase de depuracion. En segundo lugar y con este mismo objetivo se indicaran las equivalencias entre los identificadores de tarea, variable, etc. empleados en Pascal-natural y los que nos imponga el compilador.

4.3.3.1.2.- ADAPTACION DE LOS PROCEDIMIENTOS DE TRANSFERENCIA DE PARAMETROS A SUBTAREAS.-

Esta fase significara añadir algunas transferencias a variables especiales, y (o) definir algunas variables comunes a ambas tareas.

4.3.3.1.3.- DEFINICION DE LAS VARIABLES EMPLEADAS.-

Esta fase enlaza con la anterior en lo relativo a la definicion de variables, y cronologicamente debera ser la primera en llevarse a cabo, pues de la forma y el lugar donde sean definidas las variables dependera la codificacion final de las sentencias, la claridad del codigo generado y la optimizacion de los tiempos de ejecucion y ocupacion de memoria.

Evidentemente no todos los lenguajes permiten la misma flexibilidad en la definicion de variables y tipos de variables. Por ello no vamos a profundizar en el tema y simplemente indicaremos los factores a considerar.

El codigo generado sera tanto mas claro y fiable cuanto mas podamos adoptar las estructuras de datos manejadas en el programa a las estructuras previamente definidas y descritas. Ello implica en la mayoria de los casos el uso adecuado de datos de tipos matriz, registro y combinaciones de ambas estructuras.

4.3.3.1.4.- OPTIMIZACION DEL CODIGO FUENTE.-

La optimizacion del codigo esta normalmente reñida con la estructuracion del mismo, y esta afirmacion es valida incluso a nivel de estructuracion de los datos. Por esta razon los problemas de optimizacion no se plantearan hasta el final del proceso de diseño, y solo en el caso de que sea imprescindible aplicarlos, a nivel de programa fuente, pues de la estructuracion de este dependera su mantenibilidad y legibilidad.

Optimizar el codigo implicara, por ejemplo, referenciar elementos de una matriz mediante punteros que se incrementan o decrementan al acceder secuencialmente a aquella, en vez de hacerlo mediante el nombre de la matriz y el indice, evitando asi multiplicar en cada caso el indice por la longitud de cada elemento. Tambien se puede

optimizar el tiempo de ejecucion limitando al maximo los parametros a transferir entre tareas, lo cual presenta problemas de efectos secundarios incontrolados, pues llamar a un procedimiento implica que seran modificadas no solo las variables que devuelva sino tambien otras variables generales que no figuran expresamente en la sentencia de llamada. Estos efectos secundarios se pueden minimizar comentando bien la cabecera de cada tarea, pero son practicamente inevitables en sistemas de gran volumen.

Los lenguajes que permiten reducir el ambito de utilizacion de las variables definidas, facilitan la deteccion de efectos secundarios. Definir las variables en ambitos lo mas estrechos posible tiene una doble ventaja, de una parte limita los efectos secundarios potenciales y de otra optimiza la ocupacion de memoria, pues el espacio ocupado por las variables que solo se usan localmente en una tarea, puede quedar libre para otras cuando no se ejecuta esa tarea.

4.3.3.2.- LENGUAJES CON PRIMITIVAS DE CONTROL SIMPLES.-

Dentro de este calificativo se incluiran el resto de los LAN, es decir aquellos que no cuentan con primitivas de control de bucle flexibles ni (o) con sentencias condicionales abiertas. En este grupo se encuentran el FORTRAN y el BASIC, entre otros muchos.

En este caso la tarea de traduccion es algo mas

compleja, pues ademas de todas las adaptaciones mencionadas en el apartado anterior, se deben adaptar las sentencias de control del Pascal-natural a las limitadas posibilidades del LAN. En la fig.4.10 se indica una posible traduccion de las sentencias estructuradas del Pascal a las sentencias del BASIC.

En estos lenguajes se suele dar ademas el inconveniente adicional de no contar con la posibilidad de definir estructuras de datos complejas, esto es registros o combinaciones de matrices y registros. Ello supone la complicacion adicional de tener que simular estas estructuras sobre otras mas sencillas. En los casos mas complejos, en que se deberian mezclar en un mismo registro campos con distinto tipo de base puede ser imprescindible descomponer la estructura en variables independientes.

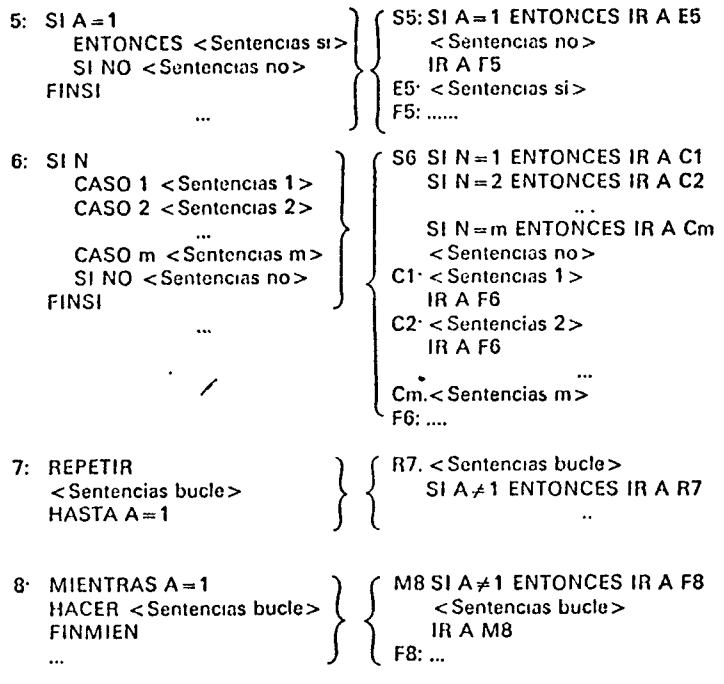


Fig. 4.10.- Ejemplo de traduccion de las sentencias de control del Pascal-natural a sentencias mas simples, del tipo SI_ENTONCES_SINO (IF_THEN_ELSE), como las empleadas en los lenguajes no estructurados.

4.3.4.- DESCRIPCION DETALLADA SEGUN NASSI-SCHNEIDERMANN.-

Esta descripcion mas detallada se realizara para las tareas no codificadas en LAN. El objetivo de esta fase del diseño es servir de puente entre una descripcion a alto nivel y la descripcion a nivel de transferencia de registros (LTR). En los casos mas sencillos no sera necesaria su realizacion, pero en la mayoria de los casos sera de gran ayuda en la descomposicion de las condiciones, que controlan los finales de bucles y las ramificaciones del algoritmo.

El siguiente paso va a ser una descripcion en LTR, en la cual las condiciones se deberan expresar en funcion de comparaciones entre registros o bits. Por todo ello se van a emplear unos diagramas de Nassi-Schneidermann con lascondiciones de control de las sentencias de iteracion o bifurcacion descompuestas en condiciones simples. En la fig.4.11 se muestra la traduccion de algunas estructuras de Pascal-natural, con condiciones compuestas, a la forma de diagramas de Nassi- Schneidermann propuesta.

Como se puede apreciar, los diagramas empleados no tienen mayor potencia de abstraccion que los organigramas tradicionalmente empleados para representar algoritmos. En cambio tienen la ventaja de no permitir estructuras distintas de las de bucle o condicionales, puesto que imponen a todo bloque una sola entrada y una sola salida. La necesidad de descomponer condiciones de control de

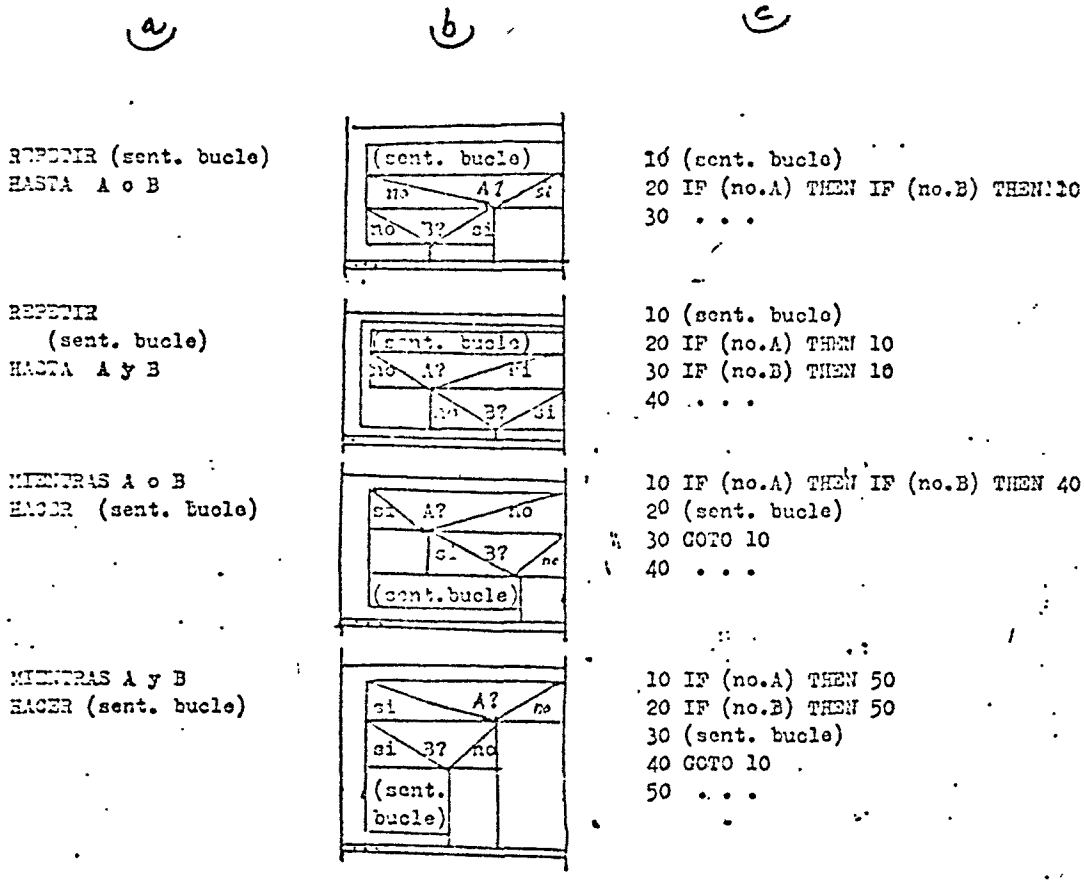


Fig. 4.11.- Equivalencia entre los diagramas de Nassi-Schneidermann (b) y las sentencias de Pascal-natural con condiciones de control compuestas (a). Estos diagramas ayudan a la descomposicion de las condiciones, para su codificacion en lenguajes no estructurados(c).

bucles en varias condiciones mas sencillas, nos obliga de hecho a sustituir las estructuras del Pascal por la estructura del bucle del PDL (apartado 2.4.1.), que permite saltar al principio o al final del bucle desde cualquier punto de este, aunque al traducirlas del Pascal queden todas agrupadas en uno de los extremos.

4.3.5.- DESCRIPCION DETALLADA DE LAS ESTRUCTURAS DE DATOS LOCALES.-

El objetivo de esta fase del diseño es doble. De una parte descomponer las estructuras de datos, adaptandolas a las limitadas posibilidades de los lenguajes de transferencia de registros (LTR). De otra parte añadir a las variables descritas a nivel general, las necesarias para almacenar resultados intermedios de las operaciones a realizar entre aquellas. A las nuevas variables les llamaremos locales, por ser utilizadas solamente dentro de una tarea, sin trascender su valor al resto de las tareas, en contra de lo que sucede con las variables generales, que sirven para transferir informacion entre tareas o para almacenar el estado de estas.

4.3.5.1.- ADAPTACION DE LAS VARIABLES GENERALES AL LTR.-

En un LTR, al igual que en algunos LAN (apartado 4.3.3.2.), solo se admiten variables simples o a lo sumo

con estructura de matriz. Para adaptar las estructuras de registro a matrices es preciso definir unas variables locales que sirvan de puntero a los distintos campos que se habian definido en el registro. Ademas se deberan desarrollar las operaciones de calculo de los indices correspondientes a cada campo o elemento de la matriz logicos, cuando la longitud de estos sea mayor que una palabra de la memoria en la que se haya definido la matriz fisica.

4.3.5.2.- VARIABLES LOCALES.-

Como ya se ha dicho estas variables serviran de puntero a estructuras o de almacen a valores intermedios de los calculos. Tambien serviran para almacenar los señalizadores que, como resultado de operaciones de relacion entre otras variables, serviran para condicionar las bifurcaciones el flujo del algoritmo.

Esta tarea de descripcion de las variables a nivel de registro se realizara en paralelo con la confeccion de los diagramas de Nassi- Schneidermann y con la descripcion en LTR, segun vayan apareciendo nuevas necesidades.

El concepto de localidad de una variable dentro de una tarea tiene la ventaja de que le asocia el caracter de temporalidad, pues tan solo estara definida mientras se este ejecutando esa tarea. Ello permite destinar una zona de almacenamiento comun para todas las variables

locales, cuyo contenido es funcion de la tarea que se esta ejecutando en ese momento. Para las tareas programadas en una maquina de uso general esta zona comun seria la formada por los registros internos de la UAL. o por la pila, o por zonas de memoria direccionables abreviadamente. Para las tareas realizadas por circuitos serian simplemente registros de proposito general accesibles desde varias tareas.