



**UNIVERSITAT AUTONOMA
DE
BARCELONA**

**FACULTAT DE CIENCIES
DEPARTAMENT D'INFORMATICA**

ADAPTACION DE LA ARQUITECTURA EN TIEMPO DE EJECUCION

MEMORIA PRESENTADA POR
JOAN SORRIBES GOMIS PARA
OPTAR AL GRADO DE DOCTOR
EN CIENCIAS (INFORMATICA)

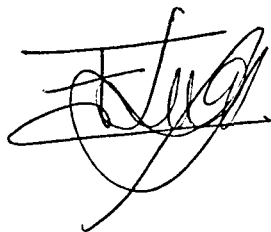
BARCELONA ABRIL 1987

ADAPTACION DE LA ARQUITECTURA EN TIEMPO DE EJECUCION

Memoria presentada por JOAN SORRIBES
GOMIS para optar al grado de Doctor
en Ciencias (Informática) por la
Universidad Autónoma de Barcelona.
Trabajo realizado en el departamento
de Informática de la Facultad de
Ciencias de la Universidad Autónoma
de Barcelona, bajo la dirección del
Dr. Emilio Luque Fadón.

Bellaterra, Abril de 1987.

Vo. Bo. Director Tesis

A handwritten signature in black ink, appearing to read 'E. Luque', with a large, sweeping flourish underneath.

Fdo. Emilio Luque

A Georgina
a Joan
i als pares

Deseo hacer constar mi agradecimiento más expresivo al Prof. Dr. Emilio Luque Fadón que con su valiosa dirección, asesoramiento y ayuda ha hecho posible la realización de esta memoria.

A la Dra. Ana Ripoll por sus acertadas sugerencias y apoyo constantes.

A Porfidio Hernández por sus comentarios alentadores y apoyo.

A Lola Isabel Rexachs por sus sugerencias y estímulo.

A Carles Siscart por el apoyo técnico proporcionado.

A los miembros de la Unidad de Control Digital por sus oportunos consejos.

Muy especialmente a mi esposa, Georgina sin cuya infinita paciencia y apoyo incondicional no hubiera sido posible este trabajo.

INDICE

PROLOGO	iv
CAPITULO I ADAPTACION DE LA ARQUITECTURA	1
I.1 INTRODUCCION	1
I.1.1 ADAPTACION FUNCIONAL DE LA ARQUITECTURA	4
1.1.a Sistemas Operativos	4
1.1.b Procesadores Definidos por un Lenguaje de Alto Nivel	6
I.1.2 ADAPTACION DE LA ARQUITECTURA ORIENTADA AL PROBLEMA	8
1.2.a Estudio Heurístico	9
1.2.b Estudio Analítico	13
I.1.3 ADAPTACION DE LA ARQUITECTURA EN TIEMPO DE EJECUCION	19
I.2 PLANTEAMIENTO DEL TRABAJO	24
I.3 ESTRUCTURA DEL SISTEMA	32
I.4 FUNCIONAMIENTO DEL SISTEMA	36
I.5 PARAMETROS DE DISEÑO	38
I.6 COMPORTAMIENTO DEL SISTEMA	40
CAPITULO II ESTRUCTURA Y FUNCIONAMIENTO DEL SISTEMA	44
II.1 ORGANIZACION Y MODO DE OPERACION DEL COPROCESADOR	44
II.1.1 FUNCIONAMIENTO DEL SISTEMA	46
II.1.2 MEMORIA DE INTERCONEXION (M. I.)	51
II.1.3 UNIDAD DE GESTION DE LA MEMORIA DE INTERCONEXION (U.G.M.I.)	62

1.3.1	ACCESO A LA MEMORIA DE INTERCO-	63
	NEXION	
1.3.1.a	Acceso en lectura	63
1.3.1.b	Acceso en escritura	65
1.3.2	ALGORITMO DE REEMPLAZAMIENTO	70
1.3.2.a	Algoritmo L.R.U.	71
1.3.2.b	Algoritmo de Distancia Máxima	74
II.1.4	UNIDAD DE CONTROL DEL COPROCESADOR	
	(U.C.C.)	78
CAPITULO III SIMULACION Y EXPERIMENTACION		81
III.1	VALORACION DE LA GANANCIA DEL METODO	
	PROPUESTO CON RESPECTO A LOS METODOS	
	CLASICOS	81
III.2	SIMULACION DEL COPROCESADOR	94
III.3	PROGRAMA SIMULADOR	101
3.1	Entrada de parámetros	104
3.2	Inicialización	108
3.3	Ejecución simulada	109
3.4	Fallo de página	110
3.5	Algoritmo de reemplazamiento	111
3.6	Creación de la interconexión	114
3.7	Actualización del algoritmo	115
3.8	El secuenciamiento	116
3.9	Visualización de resultados	116
3.10	Simulación sistemática	118
3.11	Simulación probabilística	119
III.4	INCIDENCIA DE LOS ALGORITMOS DE REEM-	
	PLAZAMIENTO EN EL COMPORTAMIENTO DEL	
	SISTEMA	125
4.1	Algoritmo de Mapping Directo	125

4.2	Algoritmo Last Recently Used (LRU)	128
4.3	Efecto de la partici3n de la M.I. en el comportamiento	133
4.4	Algoritmo de Distancia M3xima	139
4.5	Comparaci3n de los algoritmos L.R.U. y D.M.	144
III.5	EJEMPLO PARA LA COMPARACION DE LOS METODOS CLASICOS CON EL PROPUESTO	146
III.6	OBTENCION DE UN MODELO ALGORITMICO DE COMPORTAMIENTO	151
6.1	Reducci3n al lazo simple	151
6.2	Comportamiento del sistema. El modelo algoritmico	155
CAPITULO IV UTILIZACION DEL COPROCESADOR : ADAP- TACION DE LA ARQUITECTURA EN EL MICROPROCESADOR R6502		
IV.1	DESCRIPCION DEL "R6502" EN VERSION MICROPROGRAMADA	180
IV.2	INCORPORACION DEL COPROCESADOR	186
IV.3	EJEMPLO DESARROLLADO SOBRE LA MAQUINA PROPUESTA	190
CONCLUSIONES Y LINEAS ABIERTAS		195
APENDICE A	LISTADO SIMULADOR DETERMINISTICO DETERPRIM	
APENDICE B	LISTADO SIMULADOR SISTEMATICO DETSISTMI	
APENDICE C	LISTADO SIMULADOR PROBABILISTICO BRANCLRU	
APENDICE D	LISTADO MICROPROGRAMAS PARA EL R6502	
APENDICE E	LISTADO MICROPROGRAMAS DEL INTER- PRETE "J+1/2 PARA EL R6502"	
REFERENCIAS		

PROLOGO

La presente memoria describe el trabajo desarrollado en el diseño y evaluación de un coprocesador para la migración vertical dinámica en tiempo de ejecución, para sistemas microprogramados. Esta memoria ha sido estructurada en cuatro capítulos cuyos contenidos resumimos a continuación.

CAPITULO I Se presenta una introducción sobre el tema de la migración vertical, primero desde su vertiente clásica de implementación "off-line", y más tarde desde el punto de vista on-line (en tiempo de ejecución). En este capítulo, se expone en líneas generales el trabajo que ha sido desarrollado, describiendo las características básicas de la estrategia presentada, la estructura del sistema y su modo de funcionamiento, así como su simulación y aplicación a un ejemplo concreto.

CAPITULO II Este capítulo ha sido dedicado a la descripción detallada de la estructura del sistema, así como de su modo de funcionamiento, analizando cada una de las partes del coprocesador por separado, y su papel en el funcionamiento del mismo. El funcionamiento del sistema es analizado en las dos situaciones que se pueden presentar al

ejecutar una instrucción : a- que deba realizarse su migración. b- que ésta ya se encuentre migrada (versión en lenguaje intermedio o "J+1/2". Esto permite estudiar posteriormente los parámetros que influyen en la aceleración de la ejecución de un programa.

CAPITULO III Se realiza un estudio comparativo de los métodos clásicos (off-line) con el método propuesto en este trabajo (on-line). El desarrollo de un programa simulador, para un sistema que incorpore el coprocesador para la migración vertical en tiempo de ejecución, nos ha permitido analizar el comportamiento de dicho sistema y la incidencia de los parámetros fundamentales, tales como el tamaño de la Memoria de Interconexión, su partición, algoritmos de reemplazamiento, etc.. Por último, a partir de los resultados obtenidos con el programa simulador, ha sido posible elaborar un modelo algorítmico que permite la evaluación de la ganancia que se puede obtener en un sistema con el coprocesador, para una configuración de parámetros dada y un programa determinado.

CAPITULO IV La estrategia de adaptación propuesta, coprocesador para migración en tiempo de ejecución, ha sido aplicada a un ejemplo concreto, el microprocesador "R6502". Inicialmente se ha procedido a una emulación del R6502 sobre una simulación de un sistema microprogramado, para posteriormente incluir el coprocesador propuesto y generar el intérprete del lenguaje intermedio definido.

CAPITULO I

ADAPTACION DE LA ARQUITECTURA

I.1 INTRODUCCION

La eficiencia con que se puede resolver un problema con la ayuda de un ordenador, depende de la medida en que la arquitectura de dicho ordenador, soporte las primitivas que precisa el algoritmo utilizado para la resolución del problema. Dado que es imposible que la arquitectura original de una máquina de propósito general, pueda adaptarse de forma óptima a cualquier problema específico, la resolución más eficaz de cada problema implica la necesidad de una máquina de propósito especial, con una arquitectura orientada al problema a resolver. Este objetivo no se puede atacar evidentemente construyendo físicamente dicha máquina para cada problema, puesto que el coste casi nunca se podría justificar. Sin embargo, podemos abordar el problema desarrollando un método capaz de adaptar (Tuning) la arquitectura de la máquina de la forma más adecuada posible, es decir, creando la máquina virtual capaz de soportar de forma óptima

las primitivas necesarias para una tarea concreta. (Ri80)

La Adaptación de la arquitectura de un ordenador está (B185) basado en la idea general de que cualquier organismo, vivo o no, que realice alguna tarea de forma repetitiva, tiene en si mismo el germen del aprendizaje. Esta capacidad puede ser explotada al máximo con un "tuning en tiempo real", (B183) (B184) donde el ordenador aumenta su velocidad de ejecución gradualmente, como resultado de la "experiencia" que adquiere al ir encontrando lazos y estructuras repetitivas durante la ejecución de un programa.

Un mecanismo de Tuning, es un medio a través del cual la arquitectura de un sistema sigue (se adapta) constantemente las variaciones del perfil de ejecución de un programa, mejorando sus tiempos de ejecución, como consecuencia de la experiencia adquirida durante la misma.

Una primera aproximación para realizar el Tuning de la arquitectura de un ordenador a una aplicación, es a través de la Migración Vertical (Am83a) (Am83b). La Migración Vertical es una técnica muy conocida para mejorar las prestaciones de un ordenador. Con esta técnica las primitivas seleccionadas (secuencias de instrucciones o funciones) son trasladadas de un nivel a otro inferior dentro de la jerarquía de los diferentes niveles software de un ordenador, bajo la idea de aumentar la velocidad de ejecución de un sistema monoprocesador. La Migración Vertical es vista

como el modo de acelerar la ejecución de un programa, convirtiendo las primitivas seleccionadas del programa original, escrito en un lenguaje de nivel "k", en nuevas instrucciones del repertorio del lenguaje "k", creando en un nivel "j" inferior ($j < k$) las rutinas de interpretación necesarias para su ejecución. Esta técnica ha sido normalmente aplicada cuando "k" es el nivel máquina y "j" el de micro-programación.

Esta migración lo es en el sentido de que se crean las rutinas en el lenguaje de nivel "j" capaces de interpretar un conjunto de nuevas instrucciones o primitivas en el lenguaje de nivel "k", las cuales pueden incluir a otras de su mismo repertorio. De esta forma una primitiva (conjunto de segmentos o funciones) de un programa de usuario (nivel "k"), puede ser substituida por una nueva instrucción (nivel "k"), que será interpretada en el nivel "j".

El objetivo que se persigue con la Migración Vertical es, en definitiva, la síntesis de la arquitectura para su adaptación a un problema. Esta síntesis de la arquitectura puede ser realizada de dos formas : (Lu81a) (Lu83) (Ho84)

- 1 - Síntesis Funcional.
- 2 - Síntesis Orientada al Problema.

I.1.1 ADAPTACION FUNCIONAL DE LA ARQUITECTURA

En la adaptación funcional se migran aquellas funciones software que son utilizadas intensivamente en determinadas áreas de aplicación. Esta alternativa ha sido utilizada con éxito en dos áreas : **Sistemas operativos y procesadores definidos por un lenguaje de alto nivel** (Ch75) (Mi85), en las cuales la migración se lleva a cabo entre el lenguaje máquina y la microprogramación.

1.1.a Sistemas Operativos.

La migración puede ser utilizada de formas distintas para proporcionar soporte óptimo a los sistemas operativos (De84) :

- creando primitivas que serán utilizadas por un sistema operativo en microprogramación.
- creando una máquina virtual en la cual puedan ser ejecutadas las primitivas.
- microprogramando porciones del sistema operativo, parcial o enteramente.

Puesto que en una máquina microprogramable la Memoria de Control Escribible (Writable Control Store) es de tamaño limitado, no es posible migrar todas las primitivas, por lo que es necesario hacer una selección de cuales han de ser migradas. Las primitivas que son candidatas para ser microprogramadas, son aquellas demasiado específicas para su realización hardware, con muy poca probabilidad de cambiar con el tiempo y que acaparen el mayor tiempo de ejecución

del código. En (Wi69) Wirth sugería el uso de la microprogramación para la creación de instrucciones máquina especiales requeridas por los sistemas operativos, tales como las primitivas de sincronización de procesos de Dijkstra "p" y "v". La mejora de las primitivas "p" y "v" mediante la microprogramación ha sido descrita por Anceau en (An72). En (Br76) Brown enumera catorce funciones que podrían ser implementadas como primitivas mediante la microprogramación. Estas incluyen, entre otras, las siguientes :

- Primitivas de sincronización de programas.
- Mecanismos de manejo de colas.
- Conmutación de estados.

En (Br77) se describe como seleccionar las primitivas más apropiadas para su implementación así como analizar las características diferenciales entre software, hardware y firmware. Existe un modelo analítico para determinar la estrategia óptima para la implementación firmware.

Como se indica en (St78) los sistemas operativos estructurados jerárquicamente tienen una desventaja, el impacto sobre el comportamiento del sistema causado por el "overhead" de los sucesivos niveles de interpretación. La reducción de este "overhead" mediante la migración vertical se consigue alterando selectivamente dicha jerarquía de interpretación, y reimplementando las primitivas (o funciones) que la CPU usa de forma más intensiva en niveles

inferiores. Una función que ha sido migrada a un nivel inferior de la jerarquía, es ejecutada de forma más rápida y eficiente, pero también más elemental. Recientemente esta metodología ha sido aplicada al sistema operativo "UNIX" por Holtkamp (Ho85).

1.1.b Procesadores Definidos por un Lenguaje de Alto Nivel.

La ejecución de programas escritos en un lenguaje de alto nivel mediante la microprogramación, está relacionada con el soporte y gestión de las estructuras de datos incorporadas y/o definidas sobre el Lenguaje de Alto Nivel y de las estructuras de control utilizadas en programas expresados en un lenguaje particular.

El uso de la microprogramación para ejecutar programas desarrollados en un lenguaje de alto nivel puede ser visto de formas distintas :

a - Los programas pueden ser traducidos directamente a microprogramas.

b - Los programas en lenguaje de alto nivel pueden ser interpretados directamente por microprogramas.

c - Los programas pueden ser traducidos (via software o firmware) a un lenguaje intermedio, y éste a su vez ser interpretado directamente por los microprogramas.

La primera utilización conocida de la migración vertical a microprogramación, para la implementación de

procesadores de lenguajes de alto nivel, fue la realización microprogramada de Weber para EULER, sobre un sistema 360 modelo 30 de IBM (We67). Esta realización consistía de tres partes :

- Un traductor, escrito en microcódigo Modelo 30.
- Un intérprete, escrito en microcódigo Modelo 30.
- Un programa de control de entrada/salida, escrito en lenguaje máquina en el sistema 360.

Posteriormente se desarrolló una implementación microprogramada de APL para el sistema IBM 360 Modelo 25 (Ha73). En este sistema APL, el emulador APL se carga en la memoria de control. Un supervisor y un traductor se cargan en la memoria principal.

Una contribución importante en este campo lo constituyen los ordenadores Burroughs B1700 (Wi72). Estos sistemas incorporan una gran variedad de lenguajes de alto nivel, traduciendo los programas a lenguajes intermedio (lenguajes S). Existe un lenguaje S para cada lenguaje de alto nivel, y los programas en estos lenguajes S son interpretados directamente por microprogramas.

En (Lu77) se describe un sistema microprogramable a dos niveles, capaz de ejecutar directamente cualquier lenguaje de alto nivel. En este sistema, cualquiera de los lenguajes de alto nivel puede ser soportado directamente como si se tratara de lenguaje máquina. Las principales

características del lenguaje de alto nivel se soportan sobre una mezcla de hardware y firmware.

Finalmente debemos hablar del Intel iAPX 432 (In80) (My81), una máquina cuya unidad central de procesos está formada por dos chips (circuitos). El primero funciona como una unidad de decodificación de la instrucción y el segundo corresponde a una unidad de microejecución. El microcódigo sobre la arquitectura básica ejecuta un lenguaje máquina equivalente al lenguaje ADA, el cual constituye el "assembler" del iAPX 432. La traslación de las sentencias ADA a instrucciones máquina es esencialmente un proceso uno a uno.

I.1.2 ADAPTACION DE LA ARQUITECTURA ORIENTADA AL PROBLEMA

Con la adaptación orientada al problema, se pretende el reemplazamiento automático del código por microcódigo funcionalmente equivalente para programas específicos, analizados individualmente. En cuanto a la metodología a seguir para elegir cuales de las secuencias del programa del usuario deben ser migradas a microcódigo, podemos enfocarla de dos formas :

- Estudio heurístico.
- Estudio analítico.

1.2.a Estudio Heurístico.

Un primer intento de conseguir de manera automática el reemplazamiento del código, por microcódigo funcionalmente equivalente se describe en (Ab74). Este proceso heurístico puede ser subdividido en varias fases incluyendo la partición del entorno, o clasificando los problemas de acuerdo a su función, haciendo la traza de la ejecución de un programa con un "overhead" mínimo, síntesis de la arquitectura, verificación del microcódigo generado y proporcionando los cambios de la arquitectura al traductor del sistema. La mayor parte del trabajo realizado por Abd-Alla está dedicado a la fase de síntesis de la arquitectura mientras que las otras permanecen como problemas en la realización de arquitecturas de ordenadores heurísticamente adaptables.

La arquitectura especializada es sintetizada de modo iterativo, una instrucción cada vez, hasta conformar un conjunto de instrucciones con un mínimo tiempo de ejecución esperado. Cuando la mejora resultante de una iteración es menor que el "tuning overhead", la arquitectura obtenida puede considerarse óptima, y la fase de síntesis finalizada. El algoritmo de síntesis puede resumirse así (Ri80) :

- 1 - Determinación de los límites de los lazos con mayor frecuencia de ejecución en el programa.
- 2 - Asignación de los datos referenciados con mayor frecuencia en el lazo a los registros de trabajo.
- 3 - Creación de las microoperaciones de precarga de los registros de trabajo.

- 4 - Conversión del flujo de instrucciones del lazo a un flujo equivalente registro a registro.
- 5 - Traducción del flujo de instrucciones a micro-instrucciones
- 6 - Optimización del flujo de microinstrucciones.
- 7 - Creación de microoperaciones de restauración de la memoria principal.
- 8 - Eliminación de las precargas y restauraciones innecesarias.
- 9 - Parar si existe convergencia (mejora), en caso contrario regresar al primer paso.

Para demostrar la viabilidad del proceso de tuning heurístico, se utilizó sobre una rutina de mover datos y un generador de números de Fibonacci sobre una arquitectura Hewlett Packard 2100. Las mejoras en velocidad que se obtuvieron de las versiones firmware sobre las software fueron de 7.9 y 4.4 respectivamente.

El-Ayat (Ay77b) desarrollo un método heurístico que extendía la idea anterior y que consistía en los siguientes pasos :

- 1 - Traza de la ejecución y monitorización de los programas de aplicación.
- 2 - Análisis estadístico de la traza e identificación de los segmentos del código que ofrezcan una mejora significativa del comportamiento del sistema al ser migrados.

3 - Síntesis del microcódigo optimizado para migrar los segmentos identificados en el paso anterior.

El programa así migrado (adaptado), es ejecutado bajo la nueva arquitectura para comparar los resultados del programa con los obtenidos con la anterior arquitectura. Se abre un periodo de tiempo durante el cual la nueva arquitectura es analizada, y se decide si ésta es definitiva, o requiere nuevos cambios hasta obtener una solución satisfactoria al problema de aplicación atacado.

En (Ay77a) se presentan dos algoritmos para automatizar el proceso de tuning; el algoritmo de análisis y el de síntesis. La función del algoritmo de análisis es la de identificar y delimitar todos los segmentos candidatos a la migración. Esto es realizado a través de un análisis del programa de aplicación y de su perfil de ejecución. En cuanto a la síntesis, se emplearon dos algoritmos diferentes, dependiendo del tipo de segmento a ser migrado, uno para los lazos y otro para los demás casos. el segundo algoritmo, es decir para "no lazos", realiza un test para los segmentos de tamaño excesivo, con el fin de hacer una primera selección entre los posibles candidatos. Si el tamaño del segmento es adecuado, es decir, está dentro de los límites permitidos, el segmento es llevado a un "buffer" de trabajo y la primera instrucción reemplazada por una instrucción extendida. Los siguientes pasos del algoritmo, manejan expansiones individuales de instrucciones en microcódigo y síntesis de microcódigo final optimizado. El algo-

ritmo de síntesis para lazos difiere del anterior, en que utiliza registros locales de trabajo para las referencias a operandos y almacenamiento temporal de resultados intermedios en lugar de la memoria principal. Los resultados experimentales obtenidos, muestran que la ganancia en velocidad conseguida es del 35 al 40 por ciento respecto del programa original.

En (St78) se describe una metodología para sistemas estructurados jerárquicamente en capas software. Este modelo está basado en la partición de las funciones software y firmware en una estructura jerárquica, en términos de relación de "USES" : "IF P_i USES P_j THEN P_i es considerado como perteneciente a un nivel superior que P_j ".

En este método heurístico, la función a ser migrada (A) se mantiene en su propio nivel y al mismo tiempo, su versión migrada (A*) es creada en el nivel inferior. Las rutinas del nivel superior que hagan uso de "A", son ahora dirigidas a su versión migrada "A*". Si dentro esta función existe una bifurcación condicional a otra función diferente de "A", puesto que este salto no puede hacerse desde "A*" y requiere de la versión no migrada, cuando se produzca esta condición se regresará a la versión original "A" desde "A*" mediante una instrucción de Trap (llamada a una rutina de nivel superior desde un nivel inferior). La versión no migrada reejecuta la rutina desde el principio, lo cual requiere que todos los datos que han sido modificados duran-

te la ejecución de "A*" (que ha sido abandonada) deben ser restablecidos antes de entrar en "A".

Los resultados obtenidos dan un factor de 5 a 10 en la velocidad de ejecución de las primitivas en particular y del 50 por ciento en la mejora de la velocidad de ejecución del programa de aplicación.

1.2.b Estudio Analítico

Rauscher y Agrawala (Ra78) desarrollaron un método de síntesis de la arquitectura de un sistema, de forma diferente. Su enfoque enmarca el problema en los programas que son escritos en un lenguaje de alto nivel y son traducidos por un compilador a un lenguaje de representación intermedio. En lugar de la fase normal de generación de código por el compilador, se ejecuta un procedimiento que realiza las siguientes funciones :

- 1 - Generación de los microprogramas que definen una nueva arquitectura, la cual soporta de manera eficiente el programa en lenguaje de alto nivel que está siendo compilado.
- 2 - Generación del programa en lenguaje máquina que haga uso de la nueva arquitectura.

Para la definición de las nuevas instrucciones se describen dos métodos : el de "secuencia de instrucciones" y el de "estructura del programa".

En el primero se analiza la representación en lenguaje intermedio del programa, para encontrar todas las secuencias de instrucciones y el número de veces que aparece que aparece cada secuencia (Ra76). La implementación microprogramada de cada secuencia es comparada con su implementación en lenguaje máquina, y aquellas secuencias que produzcan un mayor ahorro de tiempo son elegidas como nuevas e instrucciones máquina.

El segundo método, llamado de "estructura del programa", supone un conocimiento "a priori" del comportamiento dinámico del programa, más concretamente de la frecuencias de ejecución de los bloques del programa, esto es, de las secuencias de operaciones que son ejecutadas secuencialmente sin bifurcaciones o selecciones. Para realizar el estudio dinámico del comportamiento del programa se hacen suposiciones sobre el flujo de control del mismo que permite hacer estimaciones de las frecuencias de ejecución (Ra75). Una vez conocido su comportamiento, se hacen comparaciones entre las realizaciones microprogramadas de los bloques del programa y sus versiones originales en lenguaje máquina. Finalmente se consideran las frecuencias esperadas de los bloques y aquellos bloques que generen un mayor ahorro de tiempo de ejecución se seleccionan como nuevas instrucciones máquina.

Estos dos métodos presentados por Rauscher para definir arquitecturas orientadas a aplicaciones, usan sólo

parte de la información disponible, es decir, el primer método define instrucciones que aparecen globalmente a lo largo del programa sin considerar el comportamiento dinámico del mismo, y el segundo define instrucciones que son locales a una parte del programa, no teniendo en cuenta las operaciones que constituyen los bloques a lo largo del programa. Por esta razón el propio Rauscher propone combinar ambos métodos. El nuevo método es similar al de las secuencias de instrucciones pero al que se añaden las frecuencias de ejecución de las secuencias para seleccionar las nuevas instrucciones.

En el proceso de selección de las nuevas instrucciones, Rauscher elige aquellas secuencias que producen un mayor ahorro de tiempo por cada palabra de la memoria de control. El procedimiento detallado es como sigue :

- 1 - Calcular la frecuencia esperada de ocurrencia para los bloques del programa. Llamamos V_1, V_2, \dots, V_t a los elementos de este vector.
- 2 - Encontrar todas las secuencias que aparezcan en dos o más partes del programa. Añadir todas las secuencias que constituyan bloques básicos.
- 3 - Calcular el número de apariciones ponderado para cada secuencias diferente.

$$Q_i = \sum_{j=1}^t N_{ij} V_j$$

donde N_{ij} es el número de veces que la secuencia "i"

aparece en el bloque "j".

4 - Calcular el tiempo de ahorro ponderado por espacio de la memoria de control para cada secuencia.

$$Y_i = \frac{Q_i S_i}{X_i}$$

donde X_i = número de microinstrucciones necesarias para interpretar la secuencia "i", y S_i es el tiempo ahorrado al ejecutar la secuencia "i".

5 - Repetir las siguientes operaciones hasta agotar el vector Y o llenar la memoria de control totalmente.

a- Encontrar el mayor elemento de Y (Y_i máximo) cuya versión microprogramada no supere el tamaño de la memoria de control disponible. Lo llamaremos elemento K-ésimo.

b- Añadir el microprograma de interpretación de la secuencia "K" a la memoria de control.

c- Representar todas las ocurrencias de las secuencias "K" en el programa en lenguaje intermedio por una nueva instrucción en lenguaje máquina.

d- Reevaluar N_{ij} , Q_i y Y_i para todas las secuencias que sean subsecuencias o supersecuencias de la secuencias "K".

e- Borrar el elemento K-ésimo del vector Y para eliminar posteriores consideraciones.

6 - Representar las restantes operaciones en el programa por instrucciones en lenguaje máquina que son interpretadas por microprogramas estándar en la memoria de control.

Dado que este procedimiento de síntesis de la arquitectura no considera todas las secuencias simultáneamente, pueden aparecer situaciones en las que no se den resultados óptimos.

Esta técnica fue implementada y verificada en un entorno de simulación y la mejora de comportamiento obtenida fue alrededor del 25 por ciento para los programas de aplicación seleccionados.

Una metodología analítica más rigurosa ha sido desarrollada por Luque y Ripoll (Lu81b), la cual introduce una fase de análisis dinámico.

Este proceso de adaptación comprende los siguientes pasos :

- 1 - Un análisis estático del programa (Lu80a) con objeto de identificar todas las secuencias de instrucciones que puedan ser microprogramadas. Las secuencias seleccionadas las denominaremos "secuencias microprogramables".

- 2 - Un análisis del comportamiento dinámico del programa para obtener el perfil de ejecución dado un conjunto de datos de entrada representativo..

3 - Una evaluación del tiempo de ahorro que se produciría al microprogramar cada secuencia microprogramable (Lu80b).

4 - Una selección de aquellas secuencias microprogramables que generen un mayor tiempo de ahorro durante la ejecución del programa (Lu80c). Las secuencias seleccionadas (primitivas) serán representadas por simples instrucciones de lenguaje máquina.

Estos autores plantean la resolución práctica del proceso de selección de las secuencias que seben ser migradas, dado el tamaño limitado de la memoria de control, como un problema de programación lineal binaria; el problema del Knapsack, el cual es de complejidad exponencial. Como alternativa a este método proponen la resolución de la selección mediante un algoritmo que proporciona una solución aproximada de complejidad polinómica. En (Di87) y (Lu87b) se muestran los resultados de este algoritmo aproximado.

Un enfoque similar al que acabamos de describir es el que dan Holtkamp y Wagner en (Ho84), en él también se plantea el problema de la selección de candidatos a ser microprogramados como el problema del Knapsack (complejidad exponencial) y se propone un algoritmo aproximado con el que se obtienen resultados muy próximos a los teóricos, siendo aplicado al caso particular del sistema operativo UNIX.

I.1.3 ADAPTACION DE LA ARQUITECTURA EN TIEMPO DE EJECUCION

Los mèritos esenciales de los mètodos descritos son la eliminaciòn del trabajo humano, justamente en un campo donde la habilidad y manejo de la estructura fisica de la màquina son fundamentales. Las ganancias obtenidas por los mètodos descritos fueron muy alentadoras y en muchos casos competitivas con las obtenidas por tècnicas de tuning manual. Todo ello hace pensar, creemos de forma realista, en ir hacia una automatizaciòn completa del proceso de Tuning. Para conseguir esto es preciso que el proceso de Tuning sea continuo y en tiempo real con la ejecuciòn del programa, y no una actividad aislada, previa a la ejecuciòn del programa. La optimizaciòn de un proceso global generalmente es difìcil y puede conducir a soluciones no òptimas, sin embargo tiene sentido intentarlo si examinamos los puntos que quedan por resolver. (B183)

En primer lugar, ninguno de los mètodos mencionados es completamente transparente al usuario. El Tuning se obtiene en una o mäs ejecuciones especialmente realizadas a tal efecto previamente a la ejecuciòn real. De este modo es automàtico pero no transparente.

En segundo lugar, el Tuning està basado en un perfil de ejecuciòn particular, con conjuntos de datos particulares. Esta es la crítica que Rauscher y Agrawala hacen de los mètodos heurísticos, pero su mètodo està basado en estimaciones hechas en tiempo de compilaciòn, menos especi-

fico, pero más irrealizable que las medidas hechas en tiempo de ejecución. Este problema fue también claro para El-Ayat y Howard, quienes después de hacer el Tuning de la arquitectura, proponían un periodo de observación para verificar la adaptación de la nueva arquitectura y en caso negativo rehacer el Tuning. De forma similar en el método de Luque y Ripoll, el análisis dinámico es hecho mediante la ejecución del programa para diversos conjuntos de datos de entrada, que pretenden ser representativos de la "vida" del programa, y obteniendo un promedio de frecuencias.

En tercer lugar, el perfil de ejecución de un programa puede cambiar en función del conjunto de datos de entrada. Este es el punto más débil, quizás, en la mayoría de métodos que se basan en ejecuciones preliminares o en análisis estáticos del programa. De este modo se corre el riesgo de optimizar adecuadamente en determinados periodos de tiempo, mientras que en otros puede producirse una pérdida de prestaciones (rendimiento). En los métodos clásicos tendríamos una optimización de tipo estadístico.

El objetivo que en realidad se perseguía con los métodos que se han descrito hasta ahora era el de adaptar la arquitectura de un sistema en el caso de programas que han de ser ejecutados un gran número de veces en dicho sistema, ya sean sistemas operativos o programas de aplicación, puesto que sólo en estos casos se justifican todas las fases de estudio de las secuencias microprogramables que se requieren

previamente a la ejecución de dichos programas.

En el presente trabajo, el problema que nos hemos planteado es el de conseguir dar un paso más allá de los conseguidos con los métodos hasta ahora existentes, es decir, pretendemos dotar al proceso de adaptación de la arquitectura de un sistema de características tales como (B183) :

1 - **TRANSPARENCIA** es decir, que el proceso de tuning se realice al margen del usuario, sin necesidad de que este deba llevar a cabo ningún proceso de análisis y estudio de la microprogramabilidad de las secuencias del programa; y por otra parte que sea operativo para cada ejecución del mismo.

2 - Que sea **ADAPTIVO**. Que siga constantemente el perfil de ejecución de un programa.

3 - Que se **igualmente eficiente** en medios estacionarios, quasi-estacionarios y no-estacionarios.

En general, y con objeto de enmarcar nuestro trabajo, vamos a describir la clasificación que Shin y Malek (Sh83) hacen sobre los esquemas de migración vertical, basándose en la forma de llevar a cabo los procesos de síntesis y carga del código migrado :

		SINTESIS	
		ESTATICA	DINAMICA
CARGA	ESTATICA	E/E	- -
	DINAMICA	E/D	D/D

Esquemas de Migración Vertical según Shin & Malek

El término estático se refiere al hecho de que la acción se realice previamente y al margen de la ejecución real del programa, es decir off-line. Mientras que el término dinámico supone que se realiza simultáneamente a la ejecución real del programa (on-line).

E/E : Síntesis estática, carga estática. Las secuencias candidatas son migradas y cargadas en la memoria de control, mientras exista espacio disponible para ello. Dado que el tamaño de memoria de control es limitado, no es posible migrar la totalidad del programa, por tanto se hace necesaria la selección de las secuencias candidatas a ser migradas. Este proceso de selección vendrá seguido por el de generación del código migrado, su carga en la memoria y la inclusión de la nueva instrucción sintetizada en el programa original. Todos estos procesos como ya se ha dicho son realizados como tareas previas a la ejecución del programa.

E/D : Síntesis estática, carga dinámica. Requiere que el código correspondiente a las secuencias seleccionadas, sea

migrado y almacenado con anterioridad a la ejecución, en una zona de la memoria principal destinada a ello. La carga en la memoria de control se lleva a cabo durante la ejecución del programa. Sólo cuando son necesarias para su ejecución, las partes migradas son llevadas desde la memoria principal a la de control.

D/D : Síntesis dinámica, carga dinámica. En ella las secuencias candidatas han de ser migradas y cargadas en tiempo de ejecución. Comparada con los dos esquemas anteriores, esta técnica no necesita de ningún tiempo adicional para procesos de síntesis off-line, salvo para el proceso de marcar los límites de los lazos. Este método es apropiado para el procesamiento paralelo, donde un procesador separado (coprocesador) sintetice el código migrado mientras el programa esté en ejecución. Un ejemplo correspondería al caso en que los lazos estuvieran almacenados en la memoria principal cuando se va a ejecutar el programa. Cuando el procesador inicia la ejecución del programa, el coprocesador empieza a sintetizar (migrar) lazos, buscando sus límites e identificando su localización. Las direcciones de los lazos migrados son mantenidas en una tabla y cuando la ejecución alcanza un lazo, su código migrado es cargado en la memoria de control. Sin embargo, tal y como es presentado por Shin y Malek en su trabajo, si el procesador principal intenta ejecutar una secuencia que todavía no ha sido completamente migrada por el coprocesador, deberá esperar hasta el término de su migración. Además esta técnica precisa de un conocimiento previo de los límites de las secuencias candidatas

(lazos) y su migrabilidad. Esta técnica es especialmente adecuada en medios multiprogramados en los que a cada una de las tareas de usuario es posible dedicarle el "writable control store" en su totalidad cuando es atendida por el procesador.

I.2 PLANTEAMIENTO DEL TRABAJO

La estrategia de Tuning automático que se presenta en esta memoria, se enmarca dentro del último esquema presentado (D/D), es decir, que cumple las tres características del Tuning ideal de Transparencia, Adaptación y Eficiencia, pero con las siguientes características diferenciales :

- La fase de generación del (microcódigo) código de las secuencias migradas es eliminada y substituida por la inclusión de un mecanismo de interconexión entre el código a ser migrado y el conjunto de nuevas rutinas de interpretación (nuevo intérprete).
- Funciona sin técnicas firmware de monitorización, y puede ser aplicado sin necesidad de conocer los límites de los lazos ni su migrabilidad (microprogramabilidad).
- El proceso de generación de las interconexiones es hecho en paralelo con la ejecución de la instrucción,

sin añadir ningún tiempo extra al de ejecución de un programa.

El objetivo, en definitiva, es el de reducir el tiempo de ejecución de cada programa en cada ejecución, para sistemas monoprocesador, sin realizar ningún análisis off-line del mismo, es decir, realizar un Tuning en tiempo real y completamente automático.

Para describir el sistema propuesto supongamos que estamos trabajando en una máquina con un cierto lenguaje de nivel "K" orientado al procedimiento u orientado al problema. Supongamos también que dicho lenguaje originalmente es interpretado por un nivel inferior "J", no necesariamente consecutivo con "K". Si ejecutamos un programa escrito en el lenguaje "K", todas aquellas sentencias o instrucciones pertenecientes a estructuras repetitivas dentro del programa, generarán secuencias de rutinas de interpretación, como la decodificación, el análisis sintáctico, validaciones de tipos de datos, extracción de parámetros, traslación de referencias simbólicas, etc, completamente innecesarias después de la primera iteración. Pues bien, introduzcamos en nuestra máquina una "Unidad de Adaptación" o "Coprocesador de Adaptación" (Tuning) (Lu87a) destinada a generar una traducción de las sentencias o instrucciones a un lenguaje intermedio entre los niveles "K" y "J" que podemos llamar "J+1/2". Este lenguaje "J+1/2" es interpretado en el nivel "J", pero no por el intérprete original (en "J"), sino por un nuevo intérprete, atendiendo al tipo y formato del len-

guaje intermedio introducido. Al ejecutar las sentencias correspondientes en el lenguaje "J+1/2", ya no será necesario realizar aquellas fases innecesarias por su repetitividad.

Durante la primera iteración de un lazo, y simultáneamente a la ejecución de cada sentencia o instrucción, el coprocesador genera una interconexión (sentencia o instrucción del lenguaje intermedio "J+1/2") entre la sentencia o instrucción "K" y su rutina de interpretación en el nuevo intérprete (intérprete intermedio "J+1/2"). Así durante las siguientes iteraciones del lazo, éste se ejecuta en su forma migrada (por medio de su nuevo intérprete), ahorrándonos, como consecuencia, las fases innecesarias antes mencionadas. El coprocesador trabaja en tiempo real, es decir, en paralelo con la ejecución de cada instrucción, y sin añadir ningún tiempo extra a la ejecución de un programa. De modo que si ejecutamos un programa sin ninguna estructura repetitiva, en una máquina que incorpore el coprocesador no sufre ninguna demora con respecto a su ejecución en una máquina que no posea dicho coprocesador.

Dado que el coprocesador debe trabajar en tiempo real y sin añadir tiempo a la ejecución normal, significa que cuando estamos en fase de búsqueda de una instrucción en la memoria principal, o de análisis sintáctico de una sentencia; simultáneamente el coprocesador debe determinar la existencia de su interconexión, por ello dicha interconexión

debe quedar almacenada, desde su creación, en una memoria independiente y perteneciente al coprocesador que denominamos Memoria de Interconexión (M.I.), para que este último pueda actuar sobre ella con independencia de la memoria principal (Lu86). Esta memoria de interconexión lógicamente deberá ser como mínimo igual de rápida que la principal para conocer la existencia de la interconexión lo antes posible, y nunca más tarde que la obtención de la forma original en la memoria principal. Además de la memoria de interconexión, el coprocesador incluirá una Unidad de Gestión de la Memoria de Interconexión (U.G.M.I.) y su propia Unidad de Control del Coprocesador (U.C.C.) (Fig I.1).

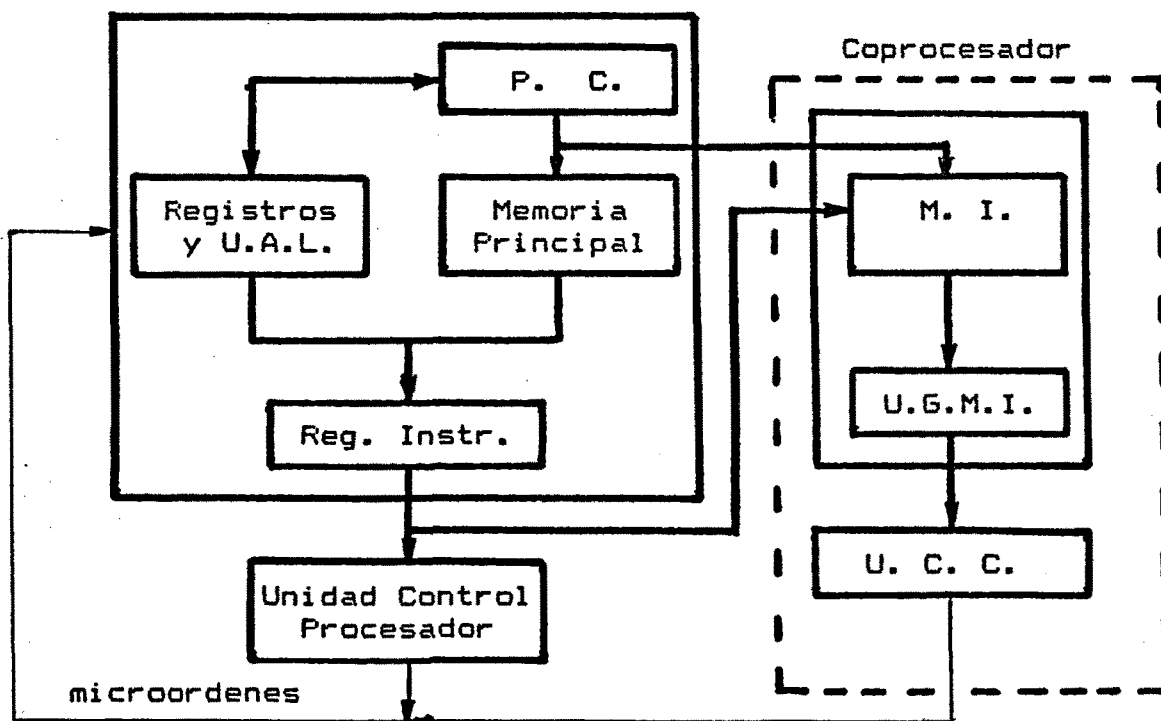


Figura I.1 Estructura del sistema para migración vertical dinámica

A diferencia de los métodos que han sido descritos con anterioridad, en los cuales los estudios estadísticos que precisan, los hacen globales en cuanto a su comportamiento y modo de realizar la adaptación sobre el perfil, así como a la hora de evaluar los rendimientos y ganancias sobre los tiempos de ejecución, el método que presentamos es óptimo en cuanto a su adaptación local, ya sea temporal o espacial, asignando dinámicamente la actuación del coprocesador en los instantes y zonas del programa que más lo precisen durante su ejecución, para obtener un mayor aumento de la velocidad de ejecución, evitando la posibilidad de que en ciertos casos, el procesador deba esperar el término de la acción del coprocesador para poder ejecutar una secuencia, como ocurre con el método propuesto por Shin y Malek (Sh83).

Se entiende por localidad espacial de un programa al hecho de que al acceder a una posición de memoria, existe una gran probabilidad de que futuros accesos nos lleven a posiciones cercanas a la actual. Este comportamiento se puede observar en la ejecución de las instrucciones, que suele ser secuencial; incluso las de salto conllevan desplazamientos generalmente cortos, esto es debido a la estructuración por bloques y/o procedimientos que definen entornos locales. En cuanto a los datos, suele ser razonable mantener próximos aquellos que tengan cierta relación lógica.

La localidad temporal se refiere a que la informa-

ción que puede ser utilizada en un futuro cercano, dentro de un programa, tiene una probabilidad alta de haber sido usada con anterioridad.

La capacidad de adaptación local del método presentado, lo hace especialmente adecuado en medios de multiprogramación, puesto que permite asignar toda la Memoria de Interconexión a cada programa, en cada una de sus ejecuciones sin que aparezca ningún "overhead". La migración vertical en sistemas multiprogramados también podría ser incorporada mediante los métodos clásicos, pero ello supondría un enorme overhead debido a las operaciones de actualización y/o restauración entre el Writable Control Store y la Memoria Principal. Una forma de reducir (pero no eliminar) este overhead, sería dividiendo el writable control store en bloques asignables a programas, pero como consecuencia de ello asignará poca capacidad de WCS a cada uno de los programas.

Justamente aprovechando la localidad de los programas durante su ejecución, podríamos ver nuestra Memoria de Interconexión como una memoria cache, la cual con un adecuado sistema de gestión, se adapte dinámicamente al perfil de ejecución. Lo que separa conceptualmente nuestra M.I. de la memoria cache, es que en M.I. no se almacena directamente código máquina sino instrucciones de lenguaje intermedio ("J+1/2"), posee una organización particular para facilitar la búsqueda de las interconexiones (la inclusión de estas se hace una a una, por palabras) y no manejamos ninguna

información relativa a los datos. Por ello nuestra M.I. podría verse como una generalización del concepto de memoria cache por cuanto la información almacenada en ella no mantiene el mismo formato ni el mismo nivel de lenguaje, y ni siquiera la misma cantidad de información; lo cual implicará un proceso diferente (más simple y rápido) de interpretación. Por otro lado en nuestra formulación la "posición conceptual" de la M.I. va ligada a la "posición" de los lenguajes implicados (dentro de la jerarquía de lenguajes de un sistema informático). Esto no implica necesariamente la creación de un nivel diferenciado (por velocidad) como en el caso de la memoria cache, sino que define un nuevo "nivel" en razón del formato adoptado y del intérprete que actuará sobre ella. Lo que si implicaría es la existencia de un almacenamiento propio para las interconexiones, para permitir la simultaneidad en el acceso a la forma codificada original y a la migrada (interconexiones).

La Memoria de Interconexión deberá ser mucho más pequeña que la memoria principal, los factores fundamentales que nos conducen a ello son :

- Al tratar de implementar un mecanismo de optimización dinámico, local y adaptivo, el recurso debe de estar en consonancia con el "tamaño" de la localidad existente en un programa (temporal y espacial), lo que implica un tamaño reducido.
- Al ser un mecanismo dinámico, local y adaptivo, un

tamaño pequeño implicará un uso intensivo del mismo produciendo un uso más eficiente del recurso.

- Su duplicidad (respecto a un nivel ya existente) o nueva incorporación (nuevo nivel más rápido) hacen necesario reducir su coste.

La desventaja que presenta un tamaño reducido y que es semejante a la que aparece en la memoria cache (Sm82), es la existencia de fallos de página (carencia de interconexión) con toda la problemática que rodea su resolución :

- Elección del tamaño total de M.I.
- Elección del número de bloques en que se particione M.I..
- Asignación de las páginas y bloques (Mapping). Entendiendo por página y bloque lo mismo que se entiende en esquemas de memoria virtual (página de memoria principal, bloque físico de M.I.)
- Localización de una referencia.
- Resolución de fallos de interconexión o fallos de página cuando la memoria está completamente llena. Algoritmo de reemplazamiento.
- Minimizar los fallos de página con un algoritmo de reemplazamiento adecuado.

La solución a todos estos puntos se consigue con

una buena partici3n de la M.I. y una adecuada U.G.M.I. que incluya un algoritmo de reemplazamiento lo m3s eficiente posible.

El tama1o de los bloques en que se particione la M.I. es uno de los par3metros fundamentales que influir3 de forma decisiva en la ganancia en velocidad del sistema. Un tama1o de bloque peque1o tiene la ventaja de que permite una mejor adaptaci3n al perfil de ejecuci3n de un programa, puesto que proporciona una partici3n m3s fina de M.I., y adem3s disminuye la probabilidad de que los bloques de M.I. contengan informaci3n que no se vaya a necesitar. La desventaja es obviamente el alto costo que representa, puesto que una partici3n fina supone un gran n1mero de bloques y ello significa replicar para cada bloque toda la circuiteria necesaria para su control. Si los bloques son grandes, disminuimos de forma importante la complejidad del hardware y en consecuencia los costes, la contrapartida la constituye una partici3n mucho m3s gruesa que no permite una adaptaci3n tan buena al perfil real de ejecuci3n de un programa

I.3 ESTRUCTURA DEL SISTEMA

El desarrollo del m3todo presentado se ha realizado para el caso particular de Adaptaci3n Din3mica, o proceso din3mico de aceleraci3n de programas, entre Lenguaje M3quina ("K") y Microprogramaci3n ("J"), pero los resultados obtenidos pueden generalizarse conceptualmente a la migra-

ción entre dos niveles de programación cualesquiera. Los motivos por los que se ha elegido el lenguaje máquina y la microprogramación para este trabajo son los siguientes :

Estos son los niveles más cercanos a la propia estructura de la máquina y por ello es posible manejar directamente los recursos de la misma. La estructura propuesta resultará más clara en cuanto a las interacciones existentes entre las diferentes partes del sistema. Así mismo la estructura del coprocesador puede ser definida a nivel de microprogramación, lo cual facilitaría su concepción en la fase experimental.

- El nuevo intérprete es escrito directamente en microprogramación, y en la mayoría de los casos bastará con introducir modificaciones a los microprogramas originales que interpretaban el lenguaje máquina. En (So82) se describe un sistema que facilita el diseño y la depuración de microprogramas, permitiendo durante su depuración actuar directamente sobre el hardware al que definitivamente irán destinados. Esta herramienta permitiría el desarrollo y depuración del nuevo intérprete

- Estos dos niveles (máquina y microprogramación) corresponden al caso más desfavorable a la hora de obtener ganancias en velocidad en una máquina, puesto que las fases de decodificación, análisis sintáctico, validaciones de tipos de datos, extracción de parámetros etc, cuyo ahorro en niveles superiores puede suponer enormes ganancias de tiempo, en estos dos niveles en concreto se reducen a las fases

de búsqueda y decodificación de la instrucción. Es por ello que con la elección de estos dos niveles tratamos de establecer la cota inferior de la ganancia en velocidad de ejecución que se puede obtener en un sistema que incorpore este método.

Partiendo pues desde ahora de un sistema microprogramado describimos cual es la estructura y funcionamiento del Tuning desarrollado en esta memoria.

Dado que los microprogramas originales de la máquina, almacenados en la memoria de control de la misma, interpretan las instrucciones máquina del repertorio del procesador, cuando se ejecute una instrucción cuya interconexión haya sido creada y por tanto ya existe, deberá ser ejecutada sobre el nuevo intérprete, el cual estará almacenado en la unidad de control del coprocesador.

El coprocesador está organizado alrededor de tres unidades básicas :

- La Memoria de Interconexión (M.I.), la cual almacena las interconexiones correspondientes a las instrucciones máquina pertenecientes a una "ventana" del programa en ejecución en la memoria principal. Esta ventana está constituida por la zona del programa donde el carácter dinámico del algoritmo ha hecho que se aplique actualmente su capacidad de adaptación local.

Dicha zona crecerá en tamaño a medida que avance la ejecución hasta encontrar una instrucción de vuelta atrás, o completar su capacidad total de almacenamiento. Cada "interconexión" es en si misma una "sentencia del lenguaje intermedio J+1/2" y ocupa una sola palabra de la M.I.. Por las razones ya expuestas, el tamaño de la M.I. es una pequeña fracción del de la memoria principal y ha sido organizada en bloques.

La Memoria de Interconexión tiene un tiempo de ciclo igual al de la memoria de control del procesador y está organizada en bloques físicos con objeto de permitir una adaptación eficiente al perfil de ejecución y facilitar el manejo de la información que constituyen las interconexiones. La memoria principal está conceptualmente organizada en páginas de igual tamaño al elegido para los bloques de la M.I.. Esto hace posible establecer una correspondencia (Mapping) entre las páginas de la memoria principal y los bloques de la M.I..

- La Unidad de Gestión de la Memoria de Interconexión (U.G.M.I.), que maneja las interconexiones y soporta el proceso de "mapping" entre las páginas de la memoria principal y los bloques de la memoria de interconexión. Para realizar esta estrategia y soportar el manejo de la estructura de bloques, se ha incorporado un algoritmo que además resuelva los fallos de página y determine que bloques han de ser reemplazados en cada

caso. Dado que la ejecución del programa avanza en incrementos de una página, la ventana de interconexiones crecerá también en incrementos de un bloque, y el algoritmo selecciona el bloque vacío a ser ocupado en cada momento. Cuando la ventana alcance su tamaño máximo permitido (el tamaño total de M.I.) y se aborda una nueva página del programa, el algoritmo selecciona el bloque que debe ser reemplazado, creando un deslizamiento dinámico (scroll) de la ventana a lo largo del programa.

- La Unidad de Control del Coprocesador (U.C.C.), que contiene los microprogramas para controlar al coprocesador y el intérprete intermedio, para que tome el control del sistema durante la ejecución de las instrucciones que tienen su correspondiente interconexión (instrucción del lenguaje intermedio) almacenada en la M.I..

I.4 FUNCIONAMIENTO DEL SISTEMA

Al iniciar la ejecución de una instrucción máquina, en su fase de búsqueda, se lanza simultáneamente una operación de lectura de la interconexión de esta instrucción en la M.I.. Todos los bloques son direccionados en paralelo para que en caso de que la interconexión estuviera en algún bloque, con independencia de cual sea, pueda ser obtenida.

En este punto hemos de distinguir las dos posibilidades, que exista la interconexión o no :

a- Si la interconexión no existe en ninguno de los bloques de la M.I., la operación de lectura que se inició en M.I. es abandonada y el sistema espera la obtención de la instrucción desde la memoria principal. Cuando la instrucción ha sido leída, el decodificador proporciona la dirección de comienzo de la fase normal de interpretación (ejecución) de la instrucción. En paralelo, se crea su interconexión (instrucción del lenguaje intermedio), escribiendo en M.I., la dirección de comienzo de la fase de interpretación sobre el nuevo intérprete, las direcciones de los operandos y todos los parámetros de la instrucción (obtenidos del registro de instrucciones). Para realizar esta escritura, el algoritmo seleccionará el bloque vacío donde deberá ser insertada la interconexión, y en caso de no existir ningún bloque libre (fallo de página) el algoritmo señalará cual debe ser reemplazado. Es necesaria la actualización del algoritmo de reemplazamiento para su posterior utilización en otros accesos. De esta manera el proceso de generación de la interconexión no añade ningún tiempo extra al tiempo normal de ejecución (Lu85).

b- Si la interconexión existe en alguno de los bloques porque la instrucción ha sido ejecutada anteriormente, la lectura iniciada en la memoria principal ya no hace falta finalizarla y es abandonada. La interconexión contiene la dirección de comienzo de la fase de interpretación interme-

dia, por lo que podemos atacar su ejecución directamente obviando las fases de búsqueda y decodificación de la instrucción. En paralelo se actualiza el algoritmo de reemplazamiento para su uso posterior.

Es importante destacar que en el caso en que la migración vertical se haga entre lenguaje máquina y microprogramación, que es el caso que nos ocupa, el tratamiento dinámico posee una ventaja adicional sobre el estático y que consiste en el hecho de que para el caso estático todos los segmentos migrados deben estar simultáneamente almacenados en el writable control store, lo cual hace que el número y tamaño de las secuencias migradas esté limitado por el tamaño de ésta. En nuestro caso se almacena en M.I. una interconexión (una palabra) por cada instrucción que sea ejecutada, por lo cual el tamaño de la M.I. sólo limita el tamaño máximo del lazo que puede capturar. En definitiva, el caso estático es global en si mismo mientras que el caso que se presenta (dinámico) es fuertemente local.

I.5 PARAMETROS DE DISEÑO

A la hora de incorporar nuestro coprocesador, en un caso real, para dotar a un sistema del mecanismo de Tuning, es preciso encontrar un compromiso entre las prestaciones adquiridas por el sistema y el coste que ello implica. Los principales parámetros que influyen en esta relación

de una forma más directa son : el tamaño total asignado a la M.I., el número de bloques en que se particione dicha memoria y el algoritmo de reemplazamiento que se elija. Por otra parte, de forma general también influirá el lenguaje intermedio definido, por las siguientes razones :

El formato de las instrucciones del lenguaje intermedio influye en el tamaño de las palabras de la M.I.

El nivel del lenguaje intermedio influirá en la estructura y tamaño del intérprete intermedio, por tanto en el tamaño de su soporte físico en el coprocesador.

También influirá en la velocidad, dependiendo de la ganancia que suponga sobre el código original.

En el caso de una partición elevada de M.I., es decir, dividirla en un gran número de bloques, aunque el tamaño de cada bloque sea pequeño (caso ideal), se consigue para el coprocesador una muy alta capacidad de adaptación al perfil de ejecución de los programas, incluso en el caso de que las secuencias lineales que conformen un lazo, se encuentren muy distantes entre si. La contrapartida que se produce ante este hecho, consiste en que al aumento del número de bloques viene acompañado de un aumento también importante del coste del sistema, debido a toda la circuitería de control que lleva asociado cada bloque, así como la necesaria para la comunicación de cada bloque con su bus.

El algoritmo de reemplazamiento debe ser implementado directamente a nivel de circuito, de este modo se consigue que la resolución de los fallos de página no supon-

gan la adición de un "overhead" al tiempo de ejecución normal del programa. Ello hace que la complejidad y por tanto el coste de dicho algoritmo crezca rápidamente con el número de bloques de la M.I.. Lo mismo ocurre con toda la lógica responsable de la localización de las interconexiones en M.I. así como con su creación y almacenamiento cuando sea preciso.

I.6 COMPORTAMIENTO DEL SISTEMA

Con objeto de realizar un análisis del comportamiento del coprocesador en operación en un sistema dado, y estudiar la incidencia de algunos de los parámetros antes mencionados sobre las prestaciones, ha sido desarrollado un programa de simulación, descrito en el capítulo tercero, el cual además nos permitirá establecer criterios para la elección de la configuración más conveniente para cada caso, estableciendo como objetivo final un modelo matemático de comportamiento del sistema con el coprocesador, para que de una forma algorítmica ser capaces de predecir el comportamiento y las ganancias en velocidad para una configuración elegida y un programa dado.

Considerando las capacidades de adaptación local del método propuesto para la realización de Tuning mediante la Migración Vertical Dinámica, es lógico pensar que el tipo de estructura que debíamos elegir para el estudio del com-

portamiento del coprocesador, y en particular analizar el del algoritmo de reemplazamiento, es la del lazo simple, o estructuras tipo lazo en general (A176), debido a que cualquier estructura repetitiva puede ser reducida al final a un simple lazo, y es en estas estructuras donde se producirán las ganancias en velocidad de ejecución. Lo que hace que el método sea localmente adaptivo al perfil de ejecución de un programa, de forma dinámica, es el hecho de que si un lazo es capturado por el coprocesador en la M.I., es decir, el lazo está totalmente traducido al lenguaje intermedio "J+1/2", la ganancia en velocidad obtenida al ser ejecutado el lazo de forma migrada, lo será con independencia de la complejidad de la estructura a la que pertenezca el lazo.

Los programas de simulación que se han desarrollado permiten analizar el comportamiento de los programas sobre un sistema que incorporase el coprocesador para migración vertical en tiempo de ejecución, midiendo parámetros tales como el número total de instrucciones ejecutadas en un programa, el número de instrucciones ejecutadas en su formato de lenguaje "J+1/2", la ganancia total en velocidad obtenida, etc. Los simuladores aceptan dos tipos de parámetros de entrada :

- Parámetros del programa :

- Descripción de la estructura estática del programa (de forma esquemática) mediante su diagrama de flujo.

- Descripción del flujo de control del programa de forma determinística o probabilística.
- Parámetros para el coprocesador :
 - Características de la M.I. : Tamaño total y número de bloques de la partición.
 - Tipo de algoritmo de reemplazamiento.

La simulación permite incorporar dos filosofías diferentes a la hora de tratar la ejecución del programa, una determinística en la cual las rupturas se secuencian en el programa, es decir las vueltas atrás en los lazos se expresan mediante el número de veces que se ejecuta dicho lazo y otra más general en que las rupturas de secuencia (no ligadas necesariamente a lazos) se expresan mediante una cierta probabilidad de salto (probabilista).

El análisis de comportamiento ha sido hecho para dos algoritmos de reemplazamiento distintos : un algoritmo L. R. U. (Last Recently Used / menos recientemente utilizado) y un algoritmo de Máxima Distancia (M.D.). El algoritmo de Máxima Distancia selecciona como bloque a ser reemplazado o reasignado, aquel, de entre todos, que en la actualidad contenga la página más alejada en el espacio de direcciones, de la que se halla en curso. Se ha comprobado que para los lazos que son capturados en su totalidad dentro de la M.I., las ganancias obtenidas con ambos algoritmos son

idénticas, pero que cuando el tamaño del lazo es superior al de la M.I. el algoritmo de L.R.U. nos dà ganacia cero y el de Màxima Distancia hace que la ganancia disminuya gradualmente al crecer el tamaño del lazo, hasta alcanzar el cero.

Para ilustrar la incorporaciòn del coprocesador en un sistema, se ha desarrollado a modo de ejemplo su inclusiòn en un sistema basado en el microprocesador R6502. Dado que dicho microprocesador està construido en un circuito integrado, de forma que no es posible acceder a las diferentes partes del mismo, se ha desarrollado la estructura de un microprocesador microprogramado y microprogramable que emule (respetando los mismos diagramas de tiempo) al R6502. Este microprocesador se ha elegido por el hecho de que existe una realizaciòn de una unidad de control orientada a la emulaciòn del R6502 (So82) (B184) y han sido desarrollados los microprogramas necesarios para la emulaciòn del R6502, a partir de los cuales se ha llevado a cabo la realizaciòn de un simulador de dicho microprocesador (He86).

CAPITULO II

ESTRUCTURA Y FUNCIONAMIENTO DEL SISTEMA

Como se ha descrito en el capítulo anterior, la implementación de la adaptación dinámica de la arquitectura de un sistema, supone la adición al mismo de un coprocesador que realice las funciones necesarias para la creación de la versión en lenguaje "J+1/2" del programa original. La estructura que se describe, corresponde al caso de adaptación de la arquitectura en el nivel de lenguaje máquina, mediante la migración a un lenguaje intermedio entre el lenguaje máquina y la microprogramación.

II.1 ORGANIZACION Y MODO DE OPERACION DEL COPROCESADOR

El coprocesador está organizado alrededor de tres unidades básicas (figura II.1) : La Memoria de Interconexión (M. I.) donde se almacenan las instrucciones del lenguaje de nivel "J+1/2", que hemos llamado "interconexiones", la Unidad de Gestión de la Memoria de Interconexión (U. G. M.

I.) que lleva a cabo la manipulación de la información almacenada en la memoria de interconexión, y la Unidad de Control del Coprocesador (U.C.C.) que proporciona el control

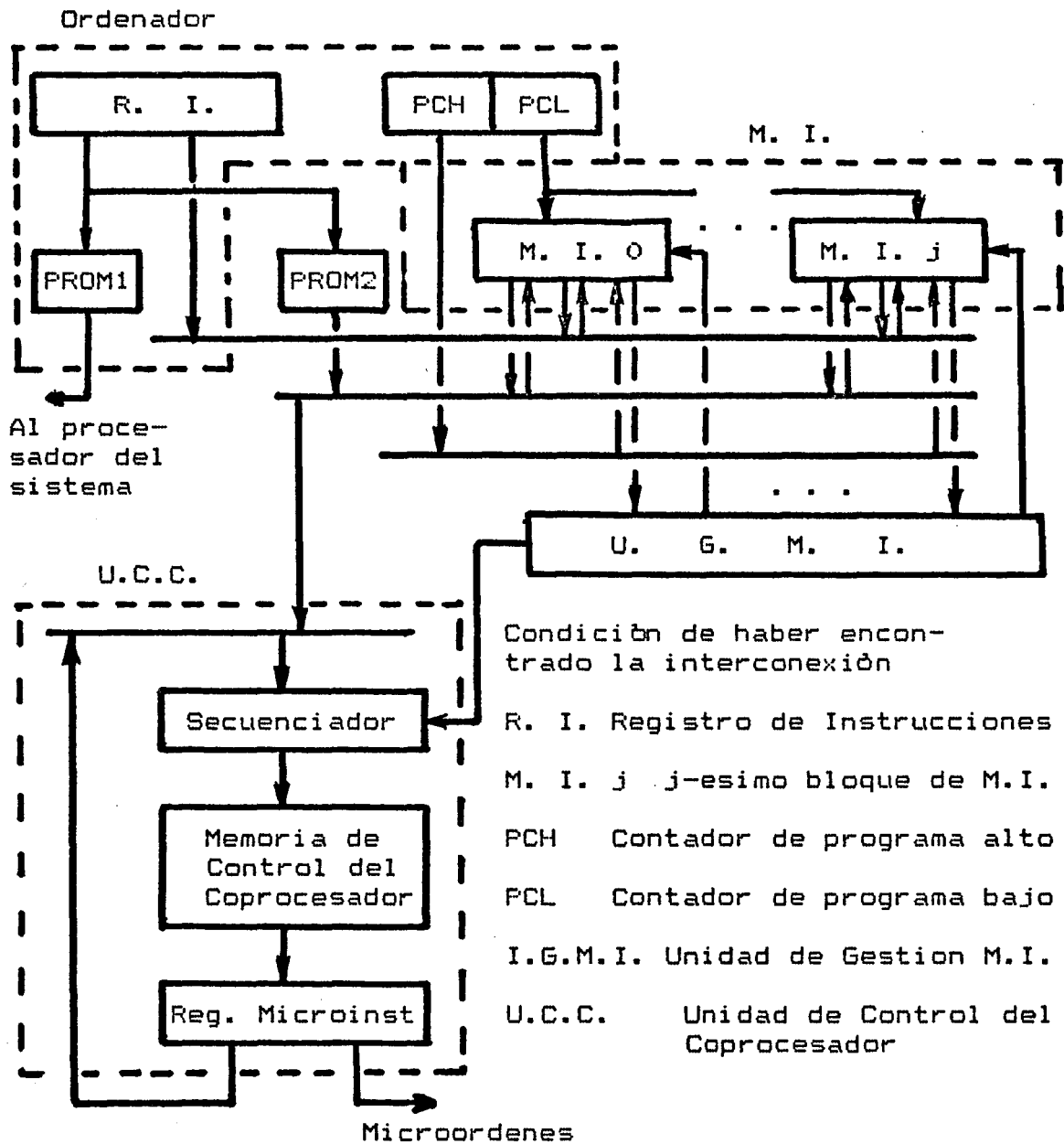


Figura II.1 Estructura del Coprocesador

de las unidades del coprocesador y almacena el intérprete del lenguaje "J+1/2". A continuación se describen detallada-

mente cada una de la unidades mencionadas.

II.1.1 FUNCIONAMIENTO DEL SISTEMA

En este apartado se intenta dar una idea de conjunto de cual es el funcionamiento global de un sistema que incorpore el método propuesto.

En la figura II.2 se muestra un diagrama general de flujo donde se recojen las principales operaciones que se realizan durante la actuación del coprocesador, en tiempo de ejecución de un programa, para la interpretación de una instrucción en su forma normal y en su forma migrada al lenguaje "J+1/2".

En primer lugar vamos a partir del supuesto de que la instrucción que se va a ejecutar, no lo ha sido nunca con anterioridad, o se ha perdido (se generó una nueva interconexión en su lugar), y que por tanto no existe en la M.I. ninguna interconexión que haga referencia a ella.

Al iniciarse la fase de búsqueda de la instrucción, se da comienzo a la lectura de la instrucción en la memoria principal, direccionándola con el contenido actual del registro contador de programa. Simultáneamente a esta lectura, se inicia otra lectura en la memoria de interconexión. Esta última lectura se realiza, en paralelo, sobre todos los bloques, direccionados al mismo tiempo por la parte baja del

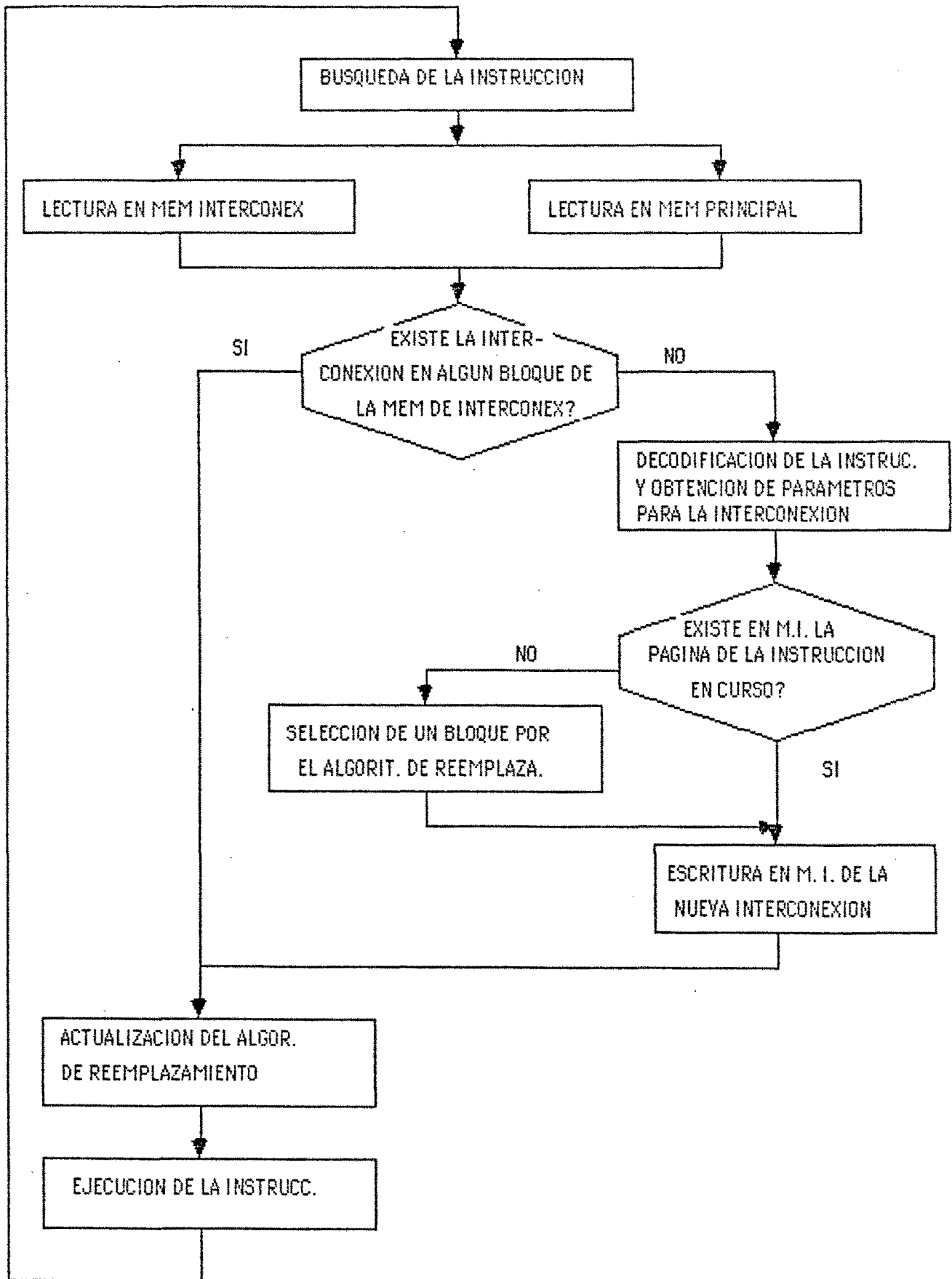


Figura II.2 Funcionamiento del Sistema

contador de programa (PCL).

Dado que se ha considerado que la interconexión de esta instrucción todavía no existe, el resultado de la lectura sobre todos los bloques de M.I. será infructuoso, debido a ser negativas todas las comparaciones realizadas entre la indicación (dentro de la interconexión) de la página de la memoria principal a la que pertenece la interconexión leída y el valor actual de la parte alta del registro contador de programa (PCH) (página en curso).

La lectura realizada sobre la M.I. es obviada y se continúa la fase de búsqueda de la instrucción que supone esperar el resultado de la lectura de la memoria principal. Una vez obtenida la instrucción, ésta será cargada en el registro de instrucciones y decodificada. Hecho esto, entraremos a la parte del microprograma correspondiente al cálculo de la dirección de los operandos.

A partir de este punto debe empezar la generación de la interconexión de esta instrucción y la búsqueda de un bloque que pueda albergarla. Para ello se analizan todos los registros de página para ver si alguno de ellos contiene la página donde se halla la instrucción en la memoria principal. Si es así, se obtiene la información que configura la interconexión y se almacena en el bloque seleccionado, en caso contrario deberemos recurrir al algoritmo de reemplazamiento.

El algoritmo de reemplazamiento seleccionará el bloque que deberá ser asignado a esta nueva página y la interconexión se almacenará en él, respetando toda la información que no corresponda a la palabra donde se inserte la interconexión. El registro de página deberá ser actualizado con el valor correspondiente a la nueva página a la que hace referencia y también el algoritmo, a su vez, habrá de ser actualizado para su uso posterior.

Una vez finalizada la ejecución normal de la instrucción, se iniciará la búsqueda de la siguiente instrucción en la memoria principal.

Supongamos ahora que la instrucción cuya fase de búsqueda está iniciándose, ha sido ejecutada con anterioridad y su interconexión se halla todavía almacenada en alguno de los bloques de la M.I..

Al iniciar la lectura de la memoria principal y de la memoria de interconexión simultáneamente conseguimos que en este caso, se obtenga el conocimiento de la existencia de la interconexión mucho tiempo antes finalizar la lectura de la memoria principal. En este momento, la lectura de la instrucción en la memoria principal es obviada y un campo del contenido de la interconexión nos proporciona la dirección de entrada al microprograma que ejecuta su versión migrada.

En los casos en que la instrucción máquina esté

constituida por más de una palabra y precise de más de un acceso a la memoria principal para completar su lectura, el tiempo ahorrado con este mecanismo se ve incrementado de forma muy notable, puesto que la interconexión tiene un formato de palabra única y en ella aparecen todos los parámetros de la instrucción.

En los casos que acabamos de mencionar, como ocurre con el microprocesador R6502 (utilizado en el ejemplo descrito en el capítulo IV), las interconexiones se almacenan en la M.I. dejando huecos entre ellas, puesto que la dirección que ocupa una interconexión en M.I. es la correspondiente a la primera palabra de la instrucción en la memoria principal. La presencia de estos huecos no genera un desaprovechamiento de la M.I. sino que estos huecos serán ocupados con gran probabilidad en subsiguientes asignaciones del bloque de M.I., dejando intacto, en ese caso, los contenidos de las interconexiones pertenecientes a asignaciones anteriores. Este comportamiento hace que en este tipo de procesadores no se produzca ningún desaprovechamiento de la M.I. sino que al contrario se aumente la probabilidad de hallar una interconexión perteneciente a antiguas asignaciones de un bloque de M.I..

Una vez ejecutada la instrucción en su versión migrada, deberá ser actualizado (si así lo requiere) el algoritmo de reemplazamiento para posteriores accesos. El siguiente paso correspondería ya al inicio de la búsqueda de

la instrucción siguiente.

A continuación se describen cada una de las unidades que componen el coprocesador en cuanto a lo que se refiere a su estructura y funcionamiento particular.

II.1.2 MEMORIA DE INTERCONEXION (M. I.)

La función básica de la Memoria de Interconexión es la de almacenar las interconexiones entre las instrucciones del programa principal y los microprogramas que las interpretan, es decir, las instrucciones del lenguaje intermedio "J+1/2". Las interconexiones están formadas por una única palabra independientemente de que el formato de las instrucciones originales sea de formato fijo o variable, de una sola palabra o de varias. Las interconexiones son generadas por el coprocesador en tiempo de ejecución del programa, y en ellas se incluyen, en una sola palabra, los parámetros a los que haga referencia la instrucción de la que son imagen.

Durante la primera ejecución de cada instrucción del programa, el coprocesador va creando, la imagen (interconexión) de la misma en la memoria de interconexión, de modo que existe una correspondencia entre las instrucciones máquina de la memoria principal y las instrucciones "J+1/2" de la memoria de interconexión. Todas aquellas instrucciones que pertenezcan a una estructura repetitiva, como puedan ser

los lazos o bucles, obtendrán su correspondiente imagen en la memoria de interconexión durante su primera ejecución, de manera que en las sucesivas iteraciones del bucle al que pertenecen, serán ejecutadas en su versión "J+1/2", incrementándose así la velocidad de ejecución del programa.

Dado que las interconexiones constituyen la alternativa a las instrucciones del programa a la hora de ser ejecutadas, el conocimiento de la existencia de una interconexión correspondiente a la instrucción en curso debe obtenerse antes de la finalización de la fase de búsqueda de dicha instrucción, es por ello que se hace necesaria la existencia de una memoria independiente de la principal (M. I.) para albergar dichas imágenes, y ser manipuladas con independencia de las instrucciones.

El tiempo de acceso de la memoria de interconexión viene delimitado por el incremento de velocidad que se quiera obtener en el sistema. Si dicho tiempo es igual al de la memoria principal, el ahorro de tiempo obtenido con la incorporación del coprocesador, es prácticamente nulo. Especialmente en aquellos sistemas en que el formato de la instrucción sea de palabra única. En el caso de procesadores con un formato de instrucción de longitud variable, o de más de una palabra, el ahorro de tiempo conseguido sería el debido al hecho de que en el formato "J+1/2" todos los parámetros de la instrucción se obtienen en un sólo acceso a M.I. y con ello nos ahorraríamos un cierto número de accesos a la memoria principal.

Si el tiempo de acceso de la M.I. es inferior al de la memoria de control, no se obtiene ninguna ganancia con respecto al caso en que dicho tiempo sea igual al de la memoria de control. Supongamos que durante el inicio de la ejecución de la última microinstrucción del micropograma que interpreta una instrucción determinada, se inicia la lectura de la siguiente instrucción en la memoria principal (como ocurriría en aquellos sistemas donde se solapa la ejecución de una instrucción con la búsqueda de la siguiente); bajo este supuesto y teniendo en cuenta, como se ha descrito en el capítulo anterior, que la lectura de la M.I. debe producirse simultáneamente con la lectura en la memoria principal, nada se ganaría con el conocimiento o no de la existencia de la interconexión de la siguiente instrucción antes de finalizar esta última microinstrucción, puesto que hasta su fin no es posible tomar ninguna decisión al respecto.

El tiempo de acceso elegido para la memoria de interconexión tiene como cota inferior al de la memoria de control del coprocesador, que será igual al de la memoria de control del procesador, puesto que más rápida no puede ser y más lenta no tendría sentido teniendo en cuenta además que los tamaños manejados para dicha M.I. serán siempre relativamente pequeños.

Dado que nuestro objetivo es el de poder capturar,

en cada momento que sea preciso, los bucles que aparezcan en el perfil de ejecución del programa, es necesario que la M.I. tenga capacidad de adaptación a ese perfil de ejecución. A la hora de analizar esta flexibilidad se nos plantean dos cuestiones :

a- A medida que avanza la ejecución del programa, independientemente de la semántica de las instrucciones y de su pertenencia o no a una estructura repetitiva, para cada instrucción se va generando su interconexión en la memoria de interconexión. El avance de la ejecución, por tanto crea una ventana lógica (conjunto de interconexiones existentes) que va siendo expandida. Este crecimiento cesará cuando en el programa se encuentre una instrucción de vuelta atrás. Esto ocurre, no por la detección de la instrucción de bifurcación sino por el hecho de que ésta cambia la secuencia del flujo de ejecución. En este caso se producirá una situación estacionaria en M.I., durante la cual transcurre la ejecución, en versión migrada, de aquellas instrucciones que pertenecen al lazo capturado. Si se alcanza el tamaño físico máximo de la M.I. y es necesario seguir avanzando, necesitamos entonces poder crear los huecos precisos para almacenar las interconexiones que se van a generar en un futuro, es decir, cuando la ventana lógica alcanza su tamaño máximo debe poder desplazarse sobre el perfil real de ejecución.

b- Una estructura repetitiva en general puede estar constituida por diferentes partes, ubicadas en la memoria de forma no necesariamente secuencial, es decir, cada una de

ellas puede estar en zonas de memoria distintas. Esto añade una nueva dificultad a la hora de capturar en M.I. dicha estructura, puesto que aún disponiendo de suficiente espacio, por su tamaño real, en la memoria de interconexión, si el espacio de direcciones que afecta es muy grande, no permitirá su inclusión total en la M.I. en un instante dado.

La solución a los dos problemas planteados se consigue mediante la organización modular en bloques de la memoria de interconexión. Esta organización, junto con un mecanismo de asignación de dichos bloques, hace posible la obtención de la flexibilidad de adaptación que antes mencionábamos, puesto que siempre será posible asignar un bloque, en el momento preciso, allí donde haga falta.

La influencia del tamaño y número de los bloques ha comportado un estudio particular, debido a la importancia que tiene sobre el comportamiento del sistema, por esta razón se analizará con mayor detalle en el siguiente capítulo.

Una vez elegido el tamaño de los bloques en que se quiere particionar la memoria de interconexión, y para facilitar el manejo de los mismos, debemos organizar, lógicamente, la memoria principal en páginas de igual tamaño al seleccionado para los bloques de M.I., de este modo se consigue una correspondencia directa entre las posiciones dentro de cada página de la memoria principal y las posi-

ciones (que llamaremos palabras) dentro de cada bloque de la memoria de interconexión. Esto permite realizar un desplazamiento (Scroll) por bloques a lo largo del perfil de ejecución del programa, si disponemos de una buena política de reemplazamiento de bloques.

Siempre es posible situar un bloque contiguo a los que constituyen en un instante dado la ventana lógica, para cubrir una nueva página del programa. Si todos se encuentran ocupados (fallo de página), será necesario desasignar alguno de los bloques, mediante un algoritmo de reemplazamiento, incorporado en la unidad de gestión de la memoria de interconexión. Este bloque será entonces asignado a la nueva página de la memoria principal donde se encuentra la ejecución del programa. De esta forma se consigue que en los bloques de M.I. se hallen almacenadas aquellas páginas de la memoria principal por las que recientemente ha transcurrido la ejecución del programa, con independencia del espacio de direcciones de la memoria principal que se vea afectado.

Supongamos que "TB" es el tamaño elegido para cada bloque, es decir que cada bloque posee TB palabras consecutivas; dado un tamaño total "TT" de la M.I. se obtendrán "NB" bloques al particionar la M.I.. La memoria principal, por tanto, se verá dividida (virtualmente) en "NP" páginas de "TB" posiciones cada una, donde $NP \gg NB$.

Para conseguir un modo fácil de acceso a las interconexiones en la M.I. y obtener un conocimiento de su

existencia o no en dicha memoria, para cada instrucción, describimos el mecanismo desarrollado :

En cualquier sistema, al iniciarse la ejecución de una instrucción, en su fase de búsqueda, la dirección de dicha instrucción en la memoria principal se encuentra contenida en el registro Contador de Programa (P. C.) (Figura II.3). Sea "LPC" la longitud en bits del registro contador de programa, dividamos éste en dos partes de longitudes : "H" y "L" ($H + L = LPC$), donde $H = \log_2 (NP)$ es la longitud de la parte alta (PCH) del contador de programa y $L = \log_2 (TB)$ la longitud de la parte baja (PCL) del mismo. De este modo la parte alta PCH señala la página donde está contenida la instrucción en curso y la parte baja PCL la posición que ocupa dentro de la página.

La salida del registro PCL es conectada, simultáneamente, a la entrada de direcciones de cada uno de los bloques de M.I.. De esta forma PCL direcciona la misma palabra en todos los bloques simultáneamente. Esta palabra ocupa, en cada bloque la misma posición que la instrucción en su página.

Para evitar que los cambios que se produzcan en el contenido del contador de programa durante la ejecución de un microprograma, pudiendo ello afectar a las direcciones que llegan a los bloques de M.I., los contenidos de PCH y PCL son cargados en dos registros auxiliares HI y LI res-

pectivamente. Estos dos últimos registros son los que atacan al bus de direcciones de los bloques de M.I. (Figura II.3).

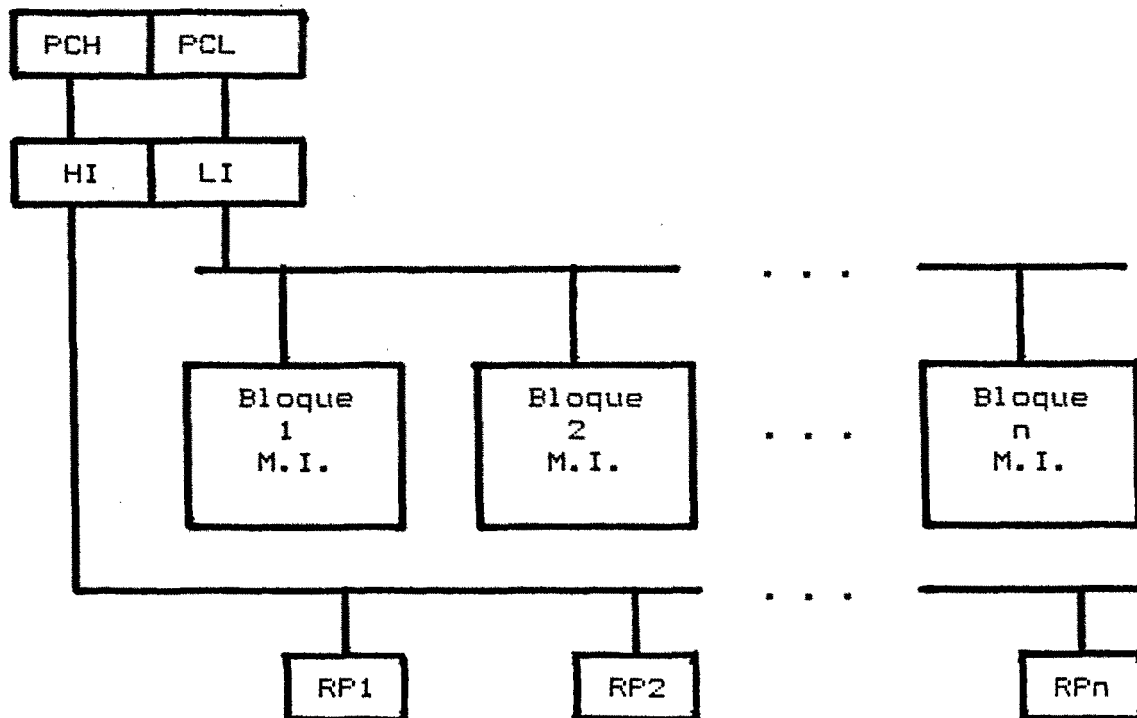


Figura II.3 Organización de la Memoria de Interconexión

Cada bloque de la memoria de interconexión tiene asociado un registro que contiene el valor de la página a la que está haciendo referencia en cada instante, por ello lo llamaremos Registro de Página (R. P.). Cada uno de estos registros deberá ser actualizado por la política de reemplazamiento que se haya elegido incorporar en la U.G.M.I.. Cada registro deberá ser cargado con el valor en curso de la parte alta del registro contador de programa (PCH), cada vez que se realice la escritura de una interconexión en el bloque al que se encuentra asociado el R.P.. La función de

estos registros es la de resolver, a un primer nivel, el emplazamiento de las interconexiones de nueva creación, antes de recurrir a la política de reemplazamiento.

El formato de la instrucción de lenguaje "J+1/2" (interconexión), almacenada en la memoria de interconexión, está organizado en tres campos (Figura II.4) :

Direcc. comienzo fase ejecución microprograma	Direccs. operandos	Direcc. página mem. princ.
Campo 1	Campo 2	Campo 3

Figura II.4 Formato de la sentencia del lenguaje intermedio "J+1/2" (Interconexión)

- El primero de los campos contiene la dirección de entrada al microprograma correspondiente para la interpretación de la instrucción, que omite ya las fases de búsqueda y decodificación de la misma, es decir, la parte donde se realiza la identificación de la instrucción. Este microprograma tiene la misma funcionalidad que el que interpreta originalmente la instrucción de la que es imagen la interconexión.

- El segundo campo contiene la dirección de los operandos, los propios operandos en el caso de direccionamiento inmediato, y todos aquellos parámetros a los que haga referencia la instrucción.

- El tercer campo contiene la página de la memoria principal donde se encuentra la instrucción de la que es imagen.

Estos tres campos constituyen la información que proporciona la interconexión entre la instrucción del programa en la memoria principal y su "ejecución directa" por el microprograma correspondiente de la memoria de control del coprocesador.

La información descrita debe ser obtenida durante la primera ejecución de la instrucción y sin añadir ningún tiempo adicional a dicha ejecución. Una vez ha sido leída la instrucción y cargada en el registro de instrucciones (R.I.), a partir del código de operación de la misma, el decodificador PROM1 proporciona la dirección de entrada a la fase de ejecución normal de la instrucción en su versión no migrada (Figura II.5) (Lu87).

Paralelamente a dicha decodificación, el decodificador perteneciente al coprocesador, PROM2, genera la dirección de entrada al punto del microprograma que deja atrás las fases de búsqueda y decodificación de la instrucción. Esta dirección será necesaria para ejecutar la instrucción, en su versión migrada, en posteriores iteraciones de la estructura a la que pertenezca. Por tanto, en dichas iteraciones se obviarán las fases mencionadas. El resultado de esta decodificación paralela (PROM2) deberá ser almacenado en la palabra correspondiente en la M.I. (Figura II.5).

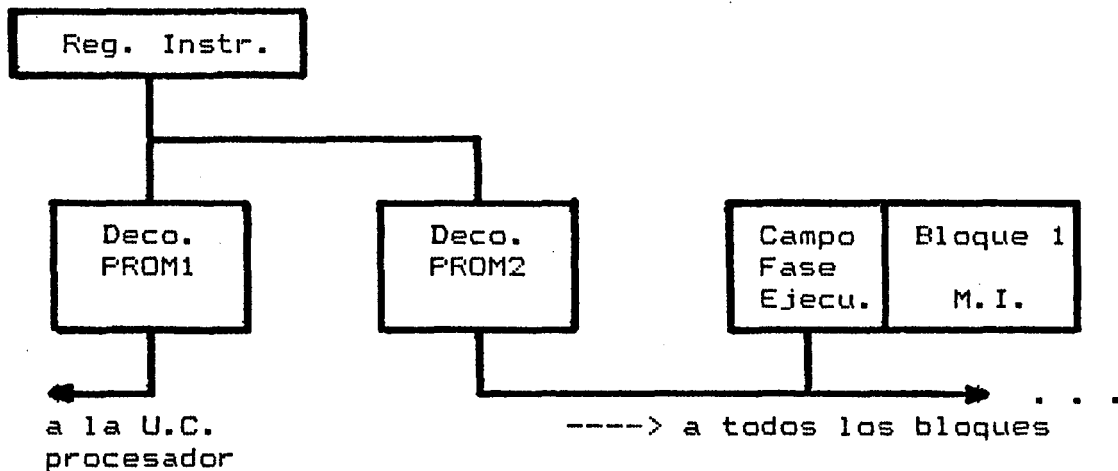


Figura II.5 Obtención de la dirección de entrada a la parte de ejecución del microprograma.

Los parámetros de la instrucción son obtenidos directamente de ésta, en el registro de instrucciones, durante su ejecución normal y almacenados en M.I. (ver Figura II.6).

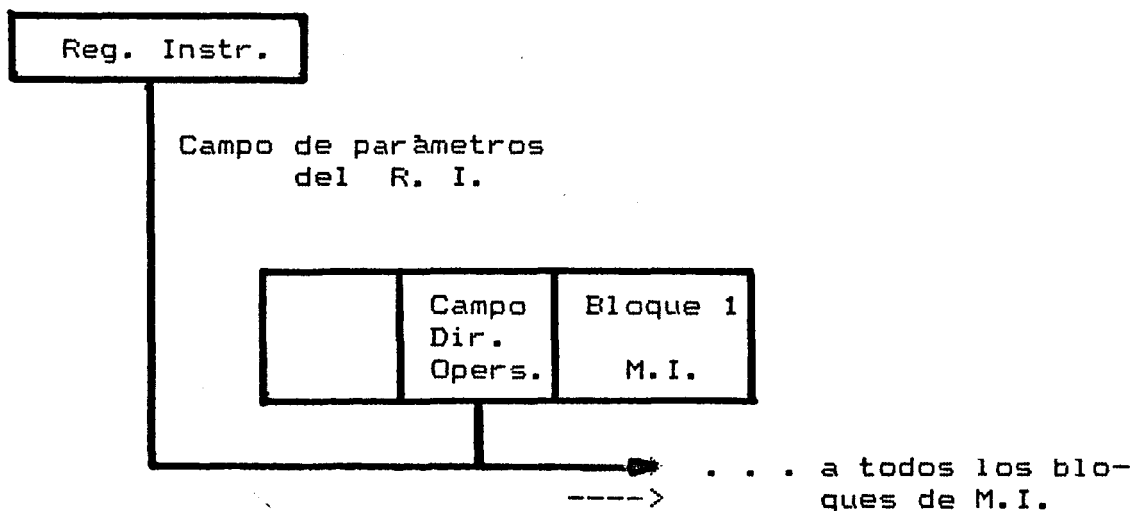


Figura II.6 Obtención de los parámetros de la instrucción

Para la obtención de estas direcciones durante la ejecución de la versión "J+1/2", se leen del segundo campo siendo posteriormente utilizados para calcular la dirección efectiva de los operandos.

El contenido del tercer campo, es decir, la identificación de la página a la que pertenece la instrucción, se obtiene como el contenido de la parte alta del contador de programa PCH, en el momento de iniciarse la ejecución de la instrucción, y se almacena en la M.I. (Figura II.7).

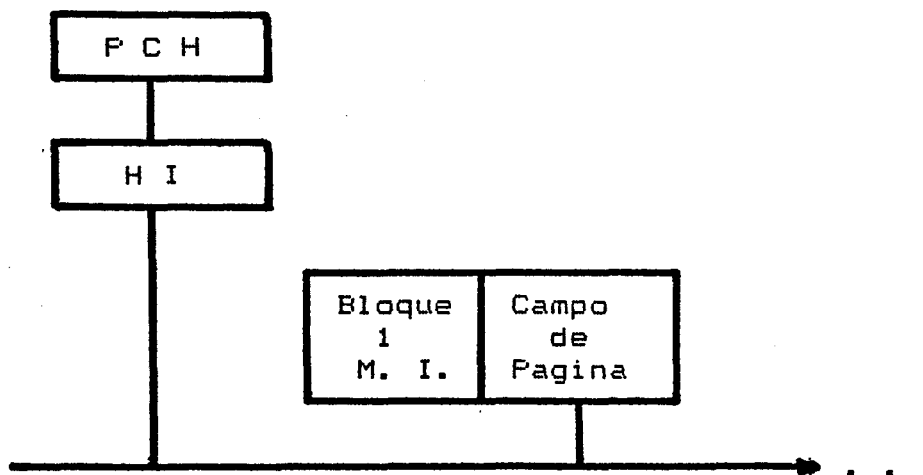


Figura II.7 Obtención de la página de la Memoria Principal a la que pertenece la interconexión.

II.1.3 UNIDAD DE GESTION DE LA MEMORIA DE INTERCONEXION (U. G. M. I.)

Esta unidad reúne toda las partes del coprocesador que tienen asignadas las funciones de manejar la infor-

mación almacenada en la M.I., tales como el acceso a la M.I. para tener conocimiento de la existencia de determinada interconexión, la selección del bloque donde deben ser escritas las nuevas interconexiones que se vayan creando y la resolución de los fallos de página.

1.3.1 ACCESO A LA MEMORIA DE INTERCONEXION

El acceso a la memoria de interconexión se produce por dos razones, conocer la existencia de una interconexión y escribir la interconexión en caso de que no exista con anterioridad.

1.3.1.a Acceso en lectura

Para conocer la existencia de una interconexión en la M.I., el mecanismo que se utiliza consiste en :

Dado que la interconexión, en caso de que exista, puede estar en cualquiera de los bloques de M.I.; previamente no tenemos ningún conocimiento de en cual de ellos puede encontrarse. Por otro lado dicha interconexión ocuparía dentro del bloque al que perteneciera, la misma posición (palabra), relativa al bloque, que la instrucción ocupe en su página, relativa a la página. Es por ello que todos los bloques deben ser direccionados simultáneamente y en paralelo, mediante la parte baja del registro contador de programa de la máquina (PCL) (Figura II.8), realizándose también en

paralelo, en todos ellos, la lectura de dicha palabra.

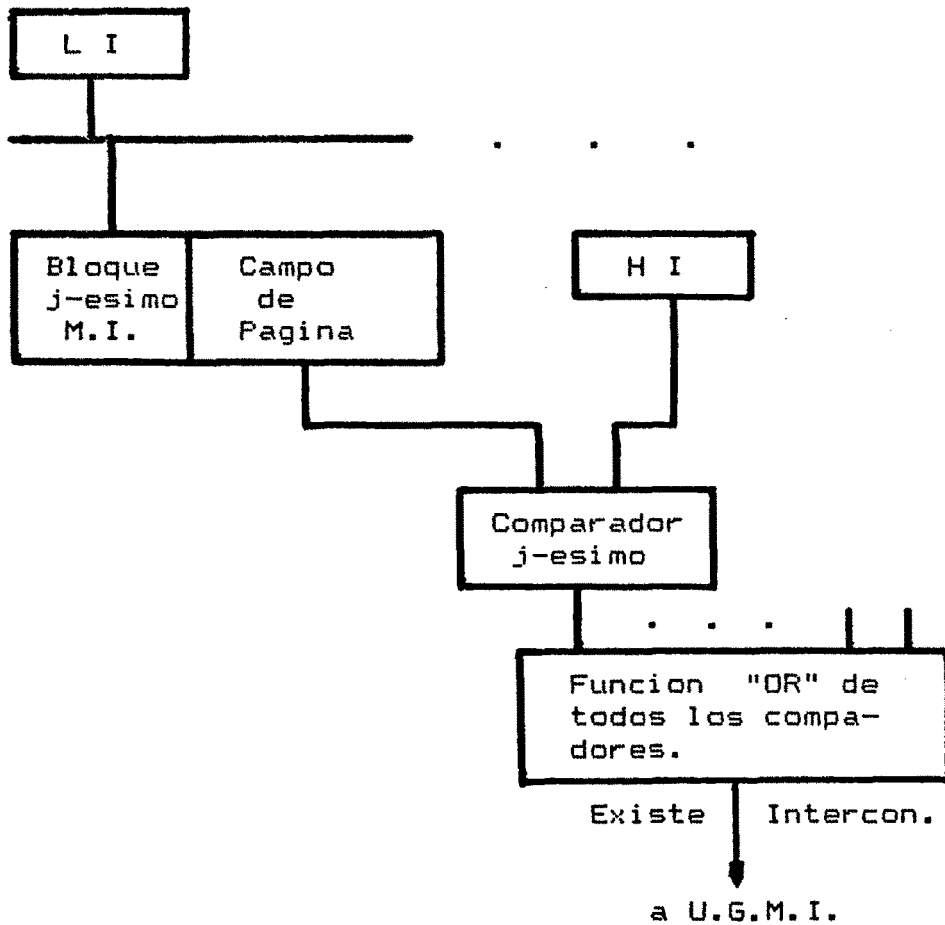


Figura II.8 Búsqueda de una interconexión en M.I.

La información leída en cada bloque (interconexiones), es en potencia la interconexión que estamos buscando. Para determinar si es así, se realiza la comparación del tercer campo de cada una de las lecturas realizadas en los bloques, con el contenido actual de la parte alta del registro contador de programa (PCH). Es decir, que simultáneamente se compara para cada bloque, la página a la que corresponde la interconexión leída con la página en curso. Si se produce la coincidencia del número de página, para

alguno de los bloques, dado que también coinciden en la posición relativa dentro de bloque y página, implica necesariamente que existe la interconexión de la instrucción cuya ejecución se está iniciando.

Una vez la U.G.M.I. tiene conocimiento de la existencia de la interconexión, indica a la Unidad de Control (del coprocesador) que la condición de existencia es verdadera y por ello la dirección de comienzo del microprograma de ejecución será proporcionada por el primer campo de la M.I..

Es importante destacar que el mecanismo utilizado para la identificación de una interconexión es independiente de su procedencia, en cuanto al bloque que la contiene. Por otro lado el método establece que si queremos obtener una interconexión en un ciclo de la memoria de control, el tiempo de acceso de la M.I. deberá ser tal que sumado con el tiempo preciso para las comparaciones sea como máximo igual al de la memoria de control.

1.3.1.b Acceso en escritura

El segundo modo en que se accede a la M.I. corresponde al caso en que debemos escribir una interconexión de nueva creación. Ello plantea diversos problemas, como son la generación de la misma, la selección de un bloque donde pueda ser insertada y la resolución de los fallos de página,

caso de que ocurran.

En el supuesto de disponer de bloque donde almacenar la interconexión, puesto que todos los bloques todavía están direccionados por la parte baja del contador de programa (PCL), bastará con poner este bloque en estado de escritura y mantener a los demás deseleccionados, para poder realizar la escritura en el bloque referido.

A la hora de elegir el bloque donde vamos a escribir la nueva interconexión se nos presentan dos posibilidades: 1- Que la página a la que pertenece la interconexión ya esté en alguno de los bloques. 2- Que no exista dicha página. Dentro de esta segunda posibilidad se plantean dos alternativas a su vez: A) Que existan uno o más bloques vacíos. B) Que todos ellos se encuentren ocupados por las diferentes páginas por las que ya ha transcurrido la ejecución del programa.

1- El primer paso que debemos dar cuando tratamos de insertar una interconexión en un bloque de M.I., es buscar si existe algún bloque que contenga la página a la que pertenece la instrucción en curso (y así compactar la información). Esto se consigue comparando simultáneamente todos los registros de página de los bloques, con el contenido actual de la parte alta del contador de programa (PCH). Si la comparación resulta positiva para alguno de los bloques, éste será el seleccionado para almacenar la interconexión de nueva creación. Este modo de proceder proporciona un

método para organizar la información de una forma coherente en el interior de los bloques, especialmente al principio de la ejecución de un programa cuando todos los bloques están vacíos y deben ir siendo rellenos de manera ordenada. De esta forma se consigue que las instrucciones que son ejecutadas y pertenecen a posiciones consecutivas en la memoria principal, originen interconexiones que también estén almacenadas de forma consecutiva dentro de su bloque.

En este punto, se consideró la posibilidad de que cada bloque tuviera asociado más de un registro de página, y que todos los registros de página correspondientes a un bloque, estuvieran organizados en forma de memoria LIFO (pila). Así, al producirse una actualización del registro de página de la cabecera, los valores antiguos serían empujados hacia abajo de la pila.

Al preguntar sobre si algún bloque almacena una página concreta, dicha pregunta se podría hacer por niveles y así recorrer hacia atrás la historia de las páginas contenidas en los bloques. Esto tendría sentido debido al hecho de que en el tiempo, un bloque puede contener interconexiones pertenecientes a distintas páginas y coexistir en el mismo bloque interconexiones referentes a páginas diferentes. Los resultados experimentales que hemos obtenido con el simulador no han sido alentadores, en el sentido de que la inclusión de dicha estructura en pila, aporta ganancias adicionales de muy bajo orden.

2- Si no existe ningún bloque cuyo registro de página haga referencia a la página de la memoria principal en curso, será necesario buscar un bloque que se encuentre vacío. En primer lugar debemos establecer el criterio bajo el cual podemos conocer si un bloque se encuentra vacío en un momento determinado. Esto es posible conseguirlo añadiendo un bit a los registros de página que nos señale dicha situación (vacío) del bloque, e inicializando cada R.P. de cada bloque con un contenido global diferente (por ejemplo con el RESET del sistema). Haciendo esto, es posible englobar la situación de bloque vacío, en la de que todos los bloques estén llenos y se produzca un fallo de página, con el algoritmo de reemplazamiento; dado que además el hablar de bloques vacíos es una situación transitoria, que solamente se produce durante las primeras fases de ejecución de un programa, llegándose rápidamente a una situación en la que todos los bloques están llenos. En consecuencia, cuando ninguno de los registros de página de los bloques haga referencia a la página en curso, lo denominaremos fallo de página, sin distinguir si existen bloques vacíos o no. En tales casos se recurre directamente al algoritmo de reemplazamiento.

La política de reemplazamiento, independientemente del algoritmo utilizado, como se puede ver, crea la posibilidad de que en un determinado bloque, puedan coexistir interconexiones referentes a instrucciones de diferentes páginas de la memoria principal, creandose con el tiempo una

distribución no regular de la información en los bloques. Este hecho, lejos de ser un perjuicio para el funcionamiento del sistema, puede producir ciertos beneficios, pues además se involucra otro factor, que se comenta a continuación, el cual inicialmente también parece negativo y no lo es.

Ya se ha dicho que las interconexiones estaban constituidas por una única palabra de M.I.. Si nos situamos en una máquina con un repertorio de instrucciones de formato variable o cuya longitud de palabra sea de más de una palabra de la memoria principal. En este caso, parece negativo el hecho de que en la memoria de interconexión van a producirse huecos entre cada dos interconexiones, dado que éstas ocupan las mismas posiciones en el bloque que sus instrucciones respectivas en la página, y por ello existe aparentemente un notable desaprovechamiento de la M.I.. Este hecho junto con el planteado anteriormente aumentan de forma notable la probabilidad de que una interconexión almacenada en un bloque que ha sido reasignado a otra página pueda ser de nuevo encontrada en una iteración lejana en el tiempo.

Cuando un bloque es reasignado a una nueva página, la información que se hallaba almacenada en él, no es borrada. Sólo aquellas posiciones que son escritas en el bloque, por la creación de nuevas interconexiones en su nueva asignación, verán destruida la información correspondiente a la interconexión que mantenía de su antigua asignación.

Esto hace que exista una probabilidad no despre-

ciable, de que en una nueva asignación las interconexiones que se creen pasen a ocupar algunos huecos entre las interconexiones de la antigua asignación, por el hecho de la búsqueda simultánea en todos los bloques; dejando intacta parte de dicha información, es decir que se produce la coexistencia de información referente a dos o más asignaciones diferentes de ese bloque. Si por alguna razón la ejecución del programa volviera a las instrucciones pertenecientes a una página que ya estuvo asignada, podría ocurrir que encontrara todavía una parte importante de las interconexiones que dejó y con ello se mejoraría el rendimiento del sistema.

1.3.2 ALGORITMO DE REEMPLAZAMIENTO

Para resolver lo que hemos definido como fallos de página, necesitamos evidentemente de un algoritmo de reemplazamiento, el cual debe ser incorporado a la unidad de gestión de la memoria de interconexión, mediante circuitería. Así, su actuación no supondrá la adición de tiempo extra en la ejecución, pudiendo por tanto operar durante la ejecución de la versión no migrada de la instrucción.

El tipo de algoritmo que se elija para ser incorporado al sistema, influye de forma bastante determinante en el comportamiento de dicho sistema y consecuentemente en la ganancia en velocidad que se pueda obtener con él. Es por ello, del mismo modo que ocurre con la partición de la M.I.,

que esta influencia merece un estudio más detallado, el cual será tratado en el próximo capítulo.

Se han elegido dos algoritmos de reemplazamiento distintos para poder de este modo comparar su influencia sobre el sistema. Un algoritmo L. R. U. (Least Recently Used) y un algoritmo de Distancia Máxima (D. M.).

1.3.2.a Algoritmo L.R.U.

El algoritmo L.R.U. (Figura II.9) en caso de fallo de página selecciona, para ser asignado a la nueva página, aquel bloque cuyo registro de página señale la página que hace más tiempo que no ha sido utilizada (Ba80) de todas las que se hallan presentes en los bloques. El tratamiento que este algoritmo hace de los fallos de página, incluye como ya hemos dicho el caso de existir bloques vacíos. La realización física de este algoritmo se basa en un conjunto de registros organizados a modo de memoria LIFO (pila), el registro cabecera almacena la identificación del bloque que debe ser seleccionado para el reemplazamiento (figura II.9).

Supongamos que todos los bloques están llenos. Cada vez que una interconexión es encontrada en un bloque, o se hace una escritura en dicho bloque (un acceso, en general), éste deberá pasar a ocupar el registro de la cabecera del algoritmo, independientemente de la posición que ocupara en el algoritmo. Se trata del más recientemente utilizado. Los registros del algoritmo que ocupan posiciones por encima

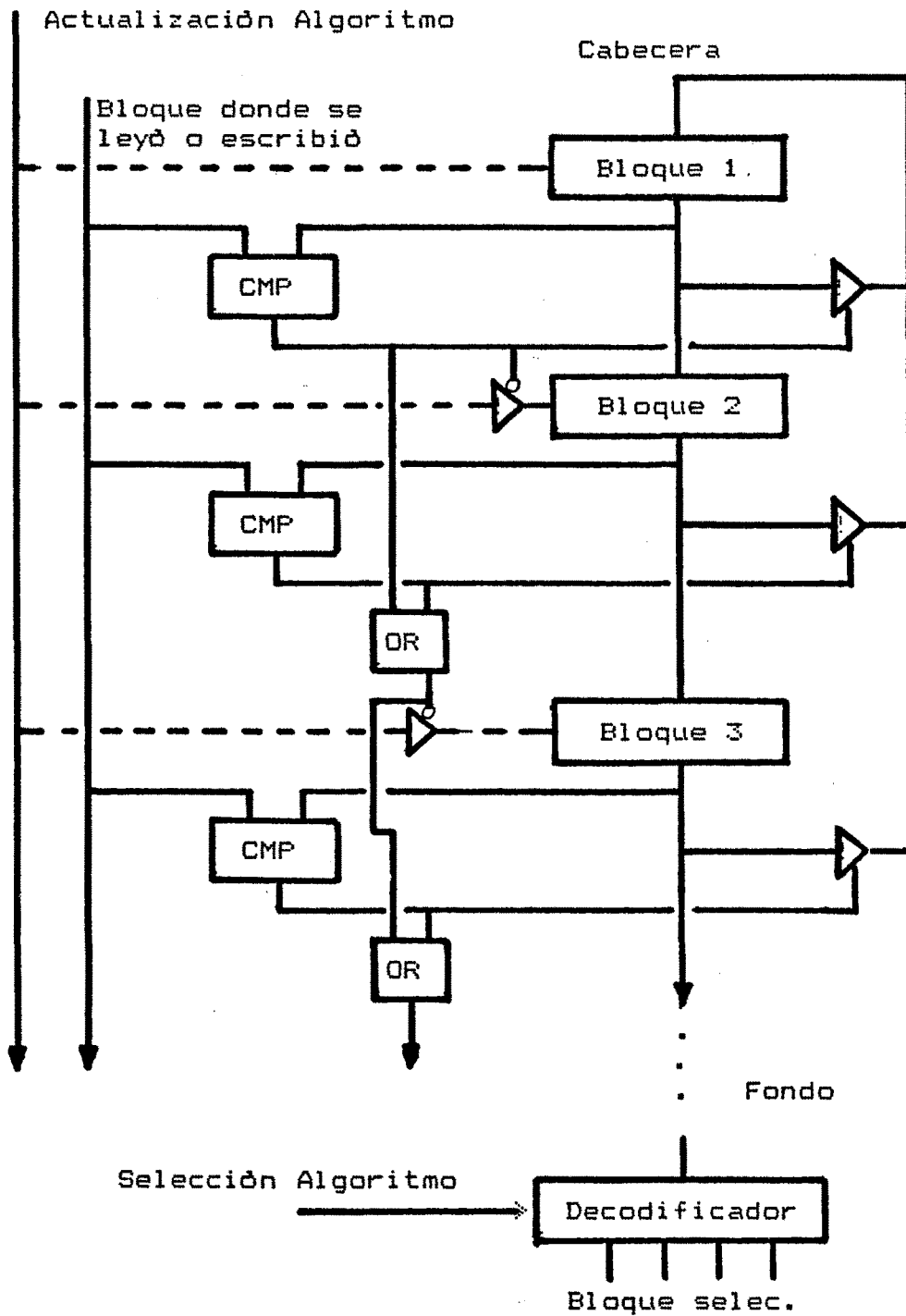


Figura II.9 Algoritmo L. R. U.

del registro cuyo contenido acaba de cargarse en la cabecera, son empujados hacia abajo para llenar este hueco, y así son aproximados al fondo. Con esta política el fondo de la pila está siempre ocupado por el bloque menos recientemente utilizado, y cuando se recurra al algoritmo de reemplazamiento para la resolución de un fallo de página, el decodificador activará la señal de capacitación del bloque correspondiente.

Al iniciarse la utilización del sistema, por ejemplo al producirse un Reset del mismo, todos los bloques de M.I. se encuentran vacíos y para indicarlo todos los registros que soportan el algoritmo deberán cargarse cada uno con un valor diferente, de modo que cada uno de los registros señale a un bloque (vacío) en concreto. Esto permitirá que el sistema pueda empezar a funcionar con todos los bloques vacíos. El registro del fondo apuntará al primer bloque (vacío) que debe ser seleccionado para escribirse en él la primera interconexión que se cree.

Cuando se realice una escritura sobre el bloque (vacío) de M.I. que señala el registro fondo del circuito propuesto, el algoritmo debe ser actualizado de forma que el bloque que acaba de ser accedido deberá pasar a ocupar el registro cabecera del algoritmo propuesto, dado que es el más recientemente utilizado. Los contenidos de todos los demás registros serán empujados hacia el fondo y así, un nuevo bloque vacío pasará a ocupar el fondo del algoritmo. De ese modo cuando haya que escribir una interconexión, y la

página en curso no se encuentre en ningún bloque, el bloque seleccionado será el del fondo de la pila de registros.

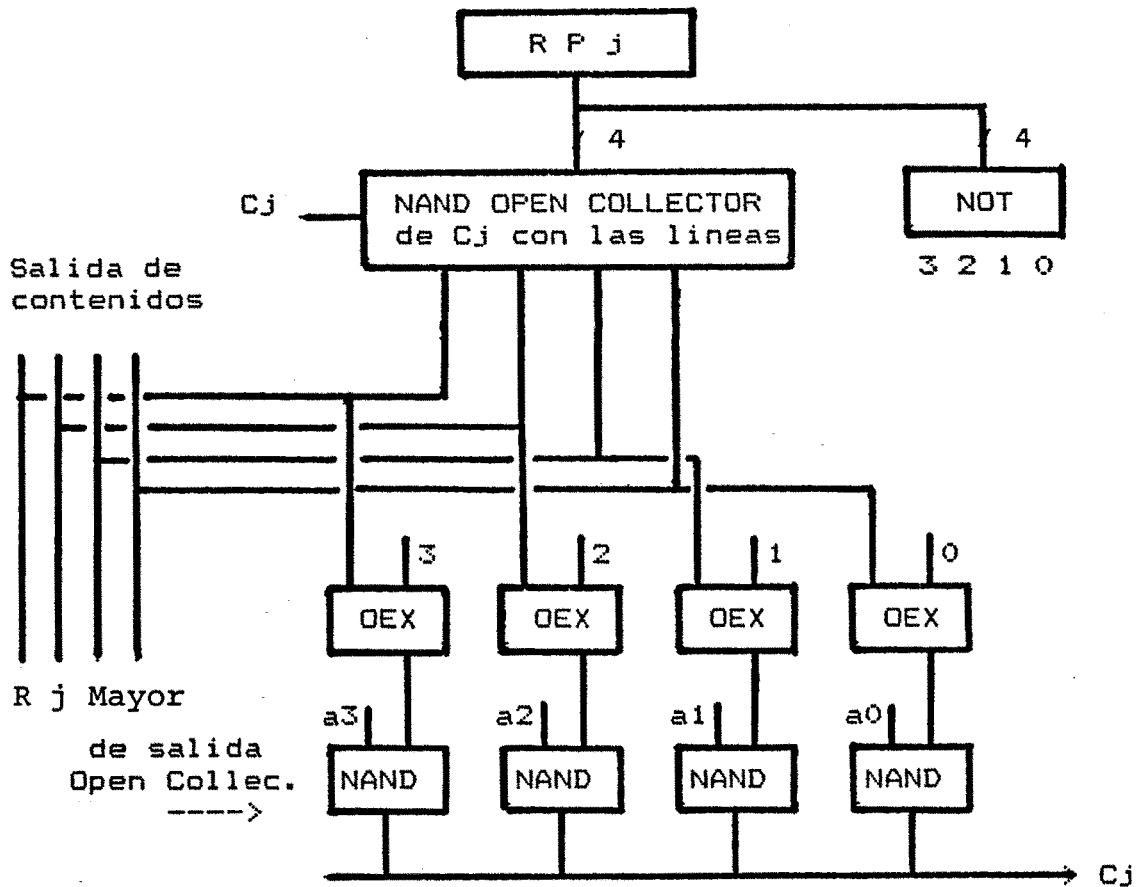
Los registros son actualizados cada vez que un bloque es seleccionado para que se escriba en él, o cada vez que en un bloque aparezca una interconexión útil. En ambos casos el bloque en cuestión pasará a ocupar la cabecera del algoritmo, y todos los demás bloques serán empujados un registro hacia el fondo. De este modo un bloque que inicialmente estuviera en la cabecera, si transcurre el tiempo y no es referenciado mientras que los demás sí, este irá acercándose al fondo del algoritmo hasta alcanzarlo y ser candidato para la siguiente asignación de página.

1.3.2.b Algoritmo de Distancia Máxima

El algoritmo de Distancia Máxima, en caso de fallo de página, selecciona como bloque a ser asignado aquel cuyo registro de página haga referencia a la página de la memoria principal más lejana, dentro del espacio de direcciones, de la página en curso. El tratamiento que este algoritmo hace de los fallos de página se basa en el hecho de que los registros de página disponen de un bit adicional que vale "1" cuando el bloque se halla vacío y como el contenido global inicial de cada registro de página es diferente, los bloques vacíos son siempre los más alejados de la página en curso.

La realización H/W del algoritmo es soportada por los registros de página, para su visualización se describe en la figura II.10 para el caso de que los registros de página tengan cuatro bit. Para calcular cual es el bloque cuyo contenido se encuentra más distante de la dirección de página en curso, es necesario determinar cuales son los bloques que contienen los valores mayor y menor de todos los presentes. Para ello mediante un un circuito secuencial que produzca las señales "aj", tantas como bits tengan los registros (hemos supuesto para el ejemplo que los registros son de cuatro bits), se efectúa el análisis bit a bit de todos los registros en paralelo (Figura II.10), desde el de más peso al de menor peso. Los contenidos de los registros se vierten al bus complementados por puertas NAND open collector, de modo que un "uno" en el bit más significativo del registro fuerza un cero en el bus.

Al iniciarse "a3", todos los registros cuyo bit más significativo no sea un "uno", son desconectados del BUS a través de su "Cj" y los que tengan un "uno" en esta posición se mantienen conectados al bus. De entre los que permanecen conectados al bus, al iniciarse "a2" se analiza del mismo modo el siguiente bit y así sucesivamente, de manera que al finalizar "a0" aquel registro que se mantiene conectado al bus es el de contenido mayor (figura II.10).



Señales secuenciales :

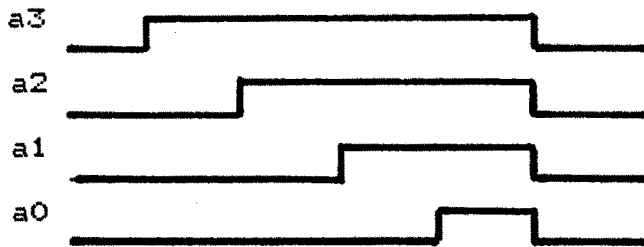


Figura II.10 Localización del RPj cuyo contenido es mayor
Algoritmo de Distancia Máxima

Para determinar el registro cuyo contenido es el menor todos, basta con sustituir las puertas NAND collector a la salida dels registro, por puertas AND open collector.

Una vez obtenidos el mayor y el menor de los contenidos, tendremos activas las señales correspondientes a

"Cj Mayor" y "Cj Menor" respectivamente. Para saber cual es el bloque que debemos seleccionar debemos diferenciar dos casos :

A) Que la página en curso se halle entre el mayor y el menor, entonces :

1 si $X - \text{menor} \geq \text{mayor} - X$ entonces Bloque = Cj menor
2 si $X - \text{menor} < \text{mayor} - X$ entonces Bloque = Cj mayor

B) Que la página en curso esté fuera del intervalo definido por "mayor y menor" :

3 si $X > \text{Mayor}$ entonces Bloque = Cj menor
4 si $X < \text{Menor}$ entonces Bloque = Cj mayor

La circuitería que resuelve estas características se describe en la figura II.11.

Los "three state" de salida controlan los "Cj" para mayor y menor, los cuales se conectan a las líneas de capacitación de los bloques y de ese modo el que se halle activo capacitará el bloque seleccionado.

Al inicializarse el sistema, será preciso que cada uno de los registros de página sea cargado con un valor diferente para cada uno de ellos, de este modo al iniciarse la ejecución de la primera instrucción de un programa, y encontrarse todos los bloques vacíos, siempre habrá un registro de página cuyo contenido sea el más alejado de la página en la que se inicia la ejecución. Una vez se ha asignado el primer bloque a la primera página que se ha

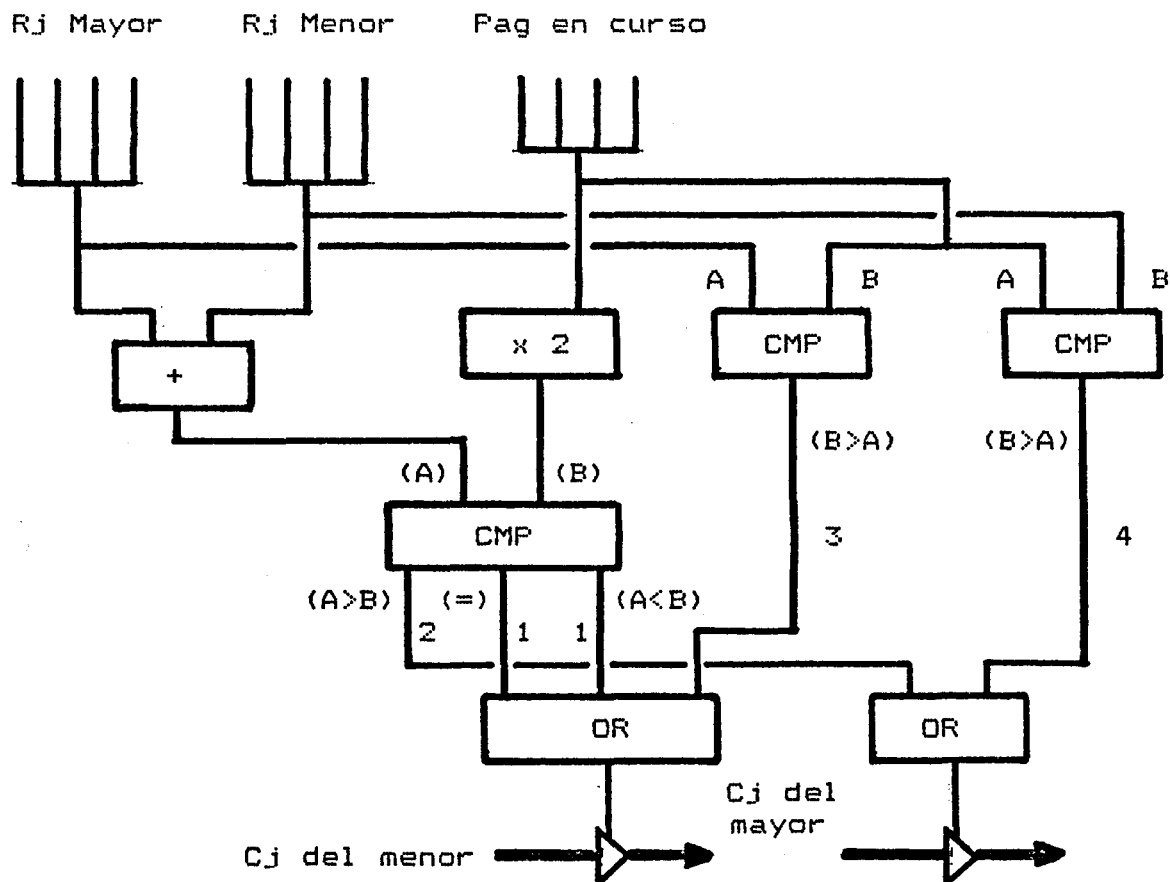


Figura II.11 Selección del bloque en escritura.
Algoritmo Distancia Máxima.

encontrado, cualquiera de los que quedan vacíos estará siempre a más distancia de la página en curso que cualquiera de los que ya se encuentran llenos.

II.1.4 UNIDAD DE CONTROL DEL COPROCESADOR (U. C. C.)

Esta es la parte del coprocesador destinada a generar todas las señales de control necesarias para el funcionamiento del coprocesador. Para una estructura

detallada de esta unidad de control del coprocesador ver (So82).

Para implementar la Unidad de Control del Coprocesador podemos hacerlo de dos formas :

A) El coprocesador tiene que funcionar en paralelo con el procesador para generar las interconexiones mientras obteniendo la información del procesador, y éste ejecuta las instrucciones en caso de que no exista su interconexión (figura II.12), o bien, en caso de que exista la interconexión deberá detener al procesador y ejecutar la instrucción en su versión "J+1/2", controlando las partes del procesador que deban actuar en cada momento. Esto supone un funciona-

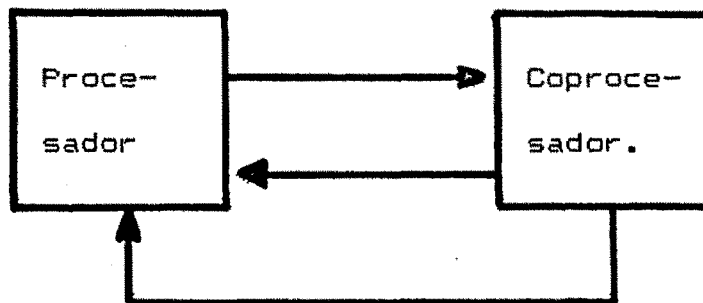


Figura II.12 Comunicación entre el procesador y el coprocesador

miento mutuo asíncrono y donde cada uno tiene su propia unidad de control por separado. La unidad de control del procesador que contiene los microprogramas del intérprete original para ejecutar las instrucciones que no hayan sido

migradas, y la unidad de control del coprocesador que controla al propio coprocesador y contiene además los microprogramas correspondientes al intérprete "J+1/2".

B) Dado que el intérprete del lenguaje "J+1/2" está escrito a nivel de microprogramación, podemos fundir la unidad de control del procesador y la del coprocesador en una sola de modo que ambos funcionen de manera mutuamente sincrónica y con el ahorro que ello puede suponer. La Unidad de Control así concebida, deberá ser modificada respecto de la original :

1) puesto que por un lado debe contener las microórdenes necesarias para controlar el funcionamiento del coprocesador.

2) por otro lado debe almacenar al intérprete "J+1/2". Para esto último es preciso :

2a) modificar los microprogramas originales por ejemplo para realizar el ciclo de búsqueda modificado.

2b) añadir nuevos microprogramas para realizar la migración así como otros alternativos para la ejecución de la forma migrada, cuando se encuentre una la interconexión de una instrucción.

Es decir la memoria de control del procesador sufrirá un crecimiento horizontal para el control del coprocesador, y vertical para el nuevo intérprete.

CAPITULO III

SIMULACION Y EXPERIMENTACION

III.1 VALORACION DE LA GANANCIA DEL METODO PROPUESTO CON RESPECTO A LOS METODOS CLASICOS

El estudio que se plantea pretende valorar los ahorros de tiempo (ganancias) obtenidos en un sistema al que se añade un coprocesador para llevar a cabo la método de generación de lenguaje intermedio, propuesto en este trabajo.

Es claro que si consideramos el caso ideal de que un programa, por su tamaño, pueda ser migrado en su totalidad, a microprogramación y almacenado en la "writable control store" de la unidad de control, los ahorros de tiempo que se obtendrían haciéndolo mediante las técnicas clásicas descritas en el primer capítulo y suponiendo que dicho programa va a ser ejecutado un número de veces suficiente que justifique el esfuerzo y el tiempo que ello supone, los ahorros de tiempo que se conseguirían serían superiores a los que se obtendrían con el método que se presenta en esta memoria.

Veamos cuales serian dichos ahorros de tiempo en ambos casos considerando el caso ideal que se ha planteado.

Dado que el tiempo necesario para la ejecución de una microinstrucción es igual al tiempo de acceso de la memoria de control, mediremos el tiempo y su ahorro por el número de microinstrucciones (o ciclos de memoria de control) contabilizadas.

A efectos solamente de nomenclatura, en este trabajo, hablaremos de migración off-line como la que hace referencia a la aplicación de las técnicas clásicas de migración de lenguaje máquina a microprogramación, en el sentido de que en dichos métodos la migración se realiza previamente a la ejecución del programa, y migración on-line como la obtenida mediante el método propuesto, puesto que en él la migración se realiza durante la ejecución del mismo.

Supongamos que la comparación que nos ocupa se ha llevado cabo en un sistema cuyo procesador tiene un repertorio de "R" instrucciones de longitud fija y palabra única. Elegimos este formato puesto que es el más desfavorable para nuestro método y no pretendemos valorar aquí los ahorros de tiempo adicionales producidos por el hecho de que en los formatos de más de una palabra, la lectura de la instrucción en la memoria principal, para extraer los parámetros, debe realizarse mediante varios accesos, mientras que con el uso del coprocesador basta un sólo acceso a la M.I. para obtener

dichos parámetros. Dado que la estructura que presenta mayores ventajas al ser migrada es la del lazo, supongamos también que vamos a ejecutar un programa constituido por un único lazo de "L" instrucciones. Sea "Pj" la probabilidad de que la instrucción "Ij" del repertorio de instrucciones, aparezca en dicho lazo ($0 \leq P_j \leq 1$). El número de .pa veces que aparecerá la instrucción Ij en el lazo, será :

$$N_j = P_j L \quad (1)$$

con

$$\sum_{j=1}^R P_j = 1 \quad (2)$$

Definimos :

Sj ≡ Número de microinstrucciones del microprograma que ejecuta la instrucción Ij en el programa original sin migrar

Cj ≡ Número de microinstrucciones del microprograma que ejecuta la instrucción Ij en el programa migrado.

Inicialmente la diferencia entre "Sj" y "Cj" está en que "Sj" incluye las microinstrucciones del microprograma que llevan a cabo la búsqueda y decodificación de la instrucción, y estas no son contabilizadas en "Cj".

Si tenemos en cuenta que en el caso off-line, es posible que los microprogramas generados hayan estado sometidos a un proceso de compactación para reducir el número

total de microinstrucciones, deberemos introducir un factor de compactación "F", el cual será considerado como el promedio estadístico que le corresponde a cada instrucción. Teniendo en cuenta este nuevo factor, C_j debe ser definido de nuevo para el caso on-line " C_{jd} " y para el caso off-line " C_{je} " de este modo :

$$C_{jd} = C_j \quad (3)$$

$$C_{je} = C_j F \quad (4)$$

Donde $0 < F \leq 1$. Si no existe compactación de los microprogramas ($F = 1$) :

$$C_{jd} = C_{je} \quad (5)$$

En primer lugar, analicemos los ahorros de tiempo que se obtienen en el caso off-line. En este caso, el ahorro de tiempo o ganancia absoluta (A_{je}) que se consigue al ejecutar una sola vez la instrucción I_j en su forma migrada es :

$$A_{je} = S_j - C_{je}$$

$$A_{je} = S_j - C_j F \quad (6)$$

y la ganancia relativa " R_{je} " vendrá dada por :

$$R_{je} = \frac{A_{je}}{S_j} = \frac{S_j - C_j F}{S_j} \quad (7)$$

con $0 \leq R_{je} \leq 1$.

La ganancia absoluta media por instrucción conse-

guida al migrar todas las instrucciones "Ij" que aparecen en el lazo, "Ame", vendrà determinada por :

$$Ame = \sum_{j=1}^R A_{je} P_j \quad (8)$$

de la ecuaci3n (6) :

$$Ame = \sum_{j=1}^R S_j P_j - \sum_{j=1}^R C_j P_j F \quad (9)$$

y la ganancia relativa media por instrucci3n, "Rme" :

$$Rme = \frac{Ame}{\sum_{j=1}^R S_j P_j} \quad (10)$$

con $0 < Rme \leq 1$. Si definimos :

$$S = \sum_{j=1}^R S_j P_j \quad (11)$$

$$C = \sum_{j=1}^R C_j P_j \quad (12)$$

Donde "S" corresponde al nùmero medio de microinstrucciones del microprograma que interpreta una instrucci3n del programa sin migrar, y "C" corresponde al mismo nùmero medio de microinstrucciones, pero para la versi3n migrada del programa, de las ecuaciones (9) y (10), las ganancias absoluta y relativa medias quedan expresadas como:

$$Ame = S - C F \quad (13)$$

$$Rme = \frac{S - C F}{S} \quad (14)$$

La ganancia absoluta total obtenida por la migración de la totalidad del lazo, "Ae", será la debida a la contribución de la migración de todas las instrucciones que intervienen :

$$Ae = Ame L \quad (15)$$

$$Ae = L (S - C F) \quad (16)$$

y la relativa total "Re" :

$$Re = \frac{Ae}{L \sum_{j=1}^R S_j P_j} \quad (17)$$

$$Re = \frac{S - C F}{S} = Rme \quad (18)$$

En el caso de que el lazo sea ejecutado un total de "V" iteraciones, la ganancia absoluta total "Ave" se expresará como :

$$Ave = V Ae \quad (19)$$

$$Ave = V L (S - F C) \quad (20)$$

y la ganancia relativa total como :

$$Rve = \frac{V Ae}{V S} = Re = \frac{S - C F}{S} \quad (21)$$

Veamos ahora las ganancias que se obtienen en el caso de incorporar el coprocesador a un sistema en el que se ejecute el mismo programa que ha sido descrito para el caso off-line. Es decir, veamos el ahorro de tiempo producido en el caso on-line.

La ganancia absoluta que se obtiene por cada vez que se migre (se genere la interconexión) la instrucción "Ij" será :

$$A_{jd} = S_j - C_{jd} \quad (C_{jd} = C_j)$$

$$A_{jd} = S_j - C_j \quad (22)$$

La ganancia absoluta correspondiente a cada instrucción "Ij" en cada una de sus apariciones en un lazo, debe ser valorada en conjunto, teniendo en cuenta el número "V" de iteraciones en el lazo, dado que el método que incorpora el coprocesador genera ganancias solo cuando hay ejecuciones iterativas de una instrucción, a diferencia de los métodos clásicos en que las ganancias se producen ya en la primera iteración.

El número total de microinstrucciones que se ejecuta durante la ejecución completa del lazo, de "V" iteraciones para cada instrucción, en su versión no migrada es "V S_j". Para calcular el número de microinstrucciones ejecutadas en el lazo con el coprocesador en operación, deben

distinguirse dos partes. Durante la primera iteración, el número de microinstrucciones ejecutadas es S_j , puesto que es durante esta primera ejecución del lazo cuando se generan las interconexiones que se utilizarán en las siguientes iteraciones, y las instrucciones se ejecutan en versión no migrada. Durante las demás iteraciones el número de microinstrucciones ejecutadas es $(V-1)C_j$, puesto que las instrucciones ya tienen su interconexión. Por ello la ganancia absoluta obtenida por instrucción es :

$$\begin{aligned} A_{jvd} &= V S_j - (S_j + (V - 1) C_j) = \\ &= (V - 1) (S_j - C_j) \end{aligned} \quad (23)$$

La ganancia absoluta media por instrucción "Amvd" que se obtiene en este caso :

$$\begin{aligned} A_{mvd} &= \sum_{j=1}^R A_{jvd} P_j \quad (24) \\ &= (V - 1) \left(\sum_{j=1}^R S_j P_j - \sum_{j=1}^R C_j P_j \right) \end{aligned}$$

de las ecuaciones (11) y (12) :

$$A_{mvd} = (V - C) (S - C) \quad (25)$$

La ganancia absoluta total "Avd" obtenida por la ejecución del lazo en sus "V" iteraciones :

$$A_{vd} = A_{mvd} L = L (V - 1) (S - C) \quad (26)$$

en cuanto a la ganancia relativa total :

$$Rvd = \frac{Avd}{L V S_j} = \frac{V - 1}{V S} (S - C) \quad (27)$$

Si comparamos los resultados obtenidos para el caso off-line :

$$Ave = V L (S - F C) \quad (20)$$

$$Rve = \frac{S - C F}{S} \quad (21)$$

con los obtenidos para el caso on-line :

$$Avd = L (V - 1) (S - C) \quad (26)$$

$$Rvd = \frac{V - 1}{V S} (S - C) \quad (27)$$

Se observa que difieren en dos factores, el número total de iteraciones y el factor de compactación. Si momentáneamente dejamos a un lado el factor de compactación, vemos que para un número de iteraciones "V" razonablemente grande (V=100) el cociente (V - 1) / V tiende a la unidad y deja de influir de manera importante. Por ejemplo para V = 20 dicho cociente vale 0.95.

El factor de compactación "F" depende de forma muy

importante de la habilidad con que el diseñador que realiza la migración off-line, compacte los microprogramas generados en la migración y por ello se hace muy difícil su cuantificación teórica. Otro hecho que influye de forma determinante en el factor de compactación, corresponde a que en los métodos clásicos (off-line) el conjunto de segmentos candidatos a ser migrados se particiona en clases de equivalencia, perteneciendo a una misma clase de equivalencia todos aquellos segmentos que puedan ser migrados al mismo microprograma. De este modo, al realizar la migración sólo se genera un microprograma para cada clase de equivalencia, incluyendo en la versión migrada, una llamada a dicho microprograma para cada uno de los elementos de la clase. En el caso on-line, se genera un conjunto de interconexiones nuevo para cada segmento de programa, aunque haya otros iguales a él; por el contrario tiene la ventaja de que no es necesaria la modificación del programa original, que se precisa en los métodos off-line.

Los métodos off-line son aplicables a medios mono-programados donde todo el "writable control store" es dedicado a los microprogramas del software en ejecución. En medios multiprogramados, es más adecuado utilizar el método on-line presentado, puesto que éste dedica todo el tamaño de la M.I. a cada uno de los programas que se ejecuten, en cualquier instante de la ejecución, mientras que los métodos off-line deberían particionar la "writable control store" para poder atender a todos los programas, con la consiguien-

te reducción de tamaño de WCS dedicado a cada uno.

El hecho de que al inicio de la ejecución del programa, en el caso off-line, la migración debe estar hecha y los microprogramas almacenados en la memoria de control, y de que el tamaño de dicha memoria es limitado, hace necesario que estos métodos precisen de un estudio estadístico de que conjunto óptimo de segmentos del programa debe ser migrado a la memoria de control. Esto no garantiza, en muchos casos, que el conjunto migrado sea el óptimo para cualquier conjunto de datos de entrada. En el caso on-line, la migración se produce durante la ejecución en función del perfil que crea el conjunto de datos de entrada que en este momento se tenga y por ello su adaptación es dinámica.

El estudio que se ha hecho para evaluar los ahorros de tiempo producidos en ambos casos (off-line y on-line) se observa que las ganancias obtenidas para el off-line son superiores a las del on-line. Dicho estudio ha sido realizado bajo el supuesto de que en el caso off-line, el programa (lazo) ha sido migrado en su totalidad a la memoria de control, pero como se acaba de decir, esto no es real, puesto que para ello necesitaríamos que el tamaño de la parte escribible de la memoria de control (W.C.S.) tuviera un tamaño igual o mayor a la suma de las longitudes de todos los microprogramas de todas las clases de equivalencia del programa, además de las llamadas a los mismos. Teniendo en cuenta que salvo compactación, cada instrucción genera un microprograma, el tamaño de la memoria de control "Tmc"

debería ser :

$$T_{mc} = L_p C F \quad (28)$$

Donde "Lp" es el número de instrucciones del programa que se migran, "C" el número medio de microinstrucciones por instrucción y "F" el factor de compactación. Esto tiene la doble dificultad del coste que supondría una memoria así y de que a la hora de dar un tamaño a la memoria de control no podemos preveer cual será la longitud media de los programas que se ejecutarán en dicha máquina. Con un tamaño de memoria de control menor que la longitud del programa, las ganancias que se obtendrán serán evidentemente menores que en el caso descrito.

En el estudio correspondiente al caso on-line, se ha tenido en cuenta que un lazo del programa es capturado en la M.I. y que el ahorro de tiempo que se produce es el calculado en (26) y (27). Esta suposición es bastante real puesto que basta con que el tamaño de la M.I. se igual a la longitud del mayor lazo del programa, para obtener un importante ahorro de tiempo, debido a que de este modo se capturarán todos los lazos de dicho programa. La longitud de los lazos de los programas de una máquina, suelen estar acotados en su mayoría entre valores bastante bien definidos, en función del repertorio de instrucciones de dicha máquina. Como se verá más adelante, en el caso on-line, es posible obtener ahorros de tiempo incluso en aquellos casos en que

el tamaño del lazo supere en cierto grado al de la M.I., debido a que la creación de las interconexiones se realiza siempre, con independencia de que su instrucción pertenezca o no a un lazo. Con una M.I. relativamente pequeña, podemos obtener ganancia en velocidad con independencia de la longitud del programa a ejecutar. Puesto que en este tipo de migración se genera una sola interconexión (palabra de M.I.) por cada instrucción que se ejecuta, el tamaño que deberá tener la M.I. será igual al espacio ocupado por el lazo más grande en la memoria principal del sistema.

En resumen, podemos decir que para conseguir las ganancias calculadas, en el caso off-line se necesita un tamaño de memoria de control igual a la longitud de todas las clases de equivalencia del programa, medida en micro-instrucciones, y para el on-line un tamaño de M.I. igual a la longitud del lazo más grande, medido en posiciones de memoria principal.

En el caso off-line no es posible la migración de segmentos que contengan llamadas a subrutinas ni otro tipo de entradas al segmento que no sea por la cabecera. En el caso on-line no existe el problema de las entradas que no sean por la cabecera y por ello se permite la migración de subrutinas.

III.2 SIMULACION DEL COPROCESADOR

Centrándonos de nuevo en el método de migración dinámica presentado, como ya se ha dicho en el primer capítulo, se ha desarrollado un programa simulador de dicho método que permite el estudio del comportamiento de un sistema que incorpore el mencionado coprocesador, analizar los parámetros fundamentales del mismo, tales como los diferentes tipos de algoritmos de reemplazamiento, tamaño total de M.I. y el número de bloques en que se particione esta, así como su incidencia sobre el comportamiento de estructuras concretas de programas.

El programa simulador ha sido realizado en Pascal y existe una versión sobre un microordenador APPLE II, y otra sobre un computador VAX-11/785. El simulador permite como entrada al mismo la definición de todos los parámetros del sistema mencionados, así como la descripción del programa a nivel de flujo del control de ejecución del mismo. El motivo por el que el programa es descrito solamente a nivel de la definición del flujo de control, es debido al hecho de que no existe interés de cual es el significado semántico de las instrucciones del programa, sino que lo realmente importante es conocer cuales son las instrucciones que se ejecutan y en que forma lo son, si en modo normal o en su versión migrada "J+1/2". Por ello nos basta saber cual es el perfil de ejecución.

Una vez se le han definido al simulador el tamaño

total de M.I. que pretendemos que tenga una configuración dada, el número de bloques en que queremos particionarla, el tipo de algoritmo de reemplazamiento elegido y la estructura del programa que debe ser simulado, el resultado que nos proporciona el simulador consiste en el número de instrucciones ejecutadas en total, el número de las que lo han sido en su forma original y el de las que han sido ejecutadas en su versión "J+1/2", calculando con ello la ganancia obtenida en la ejecución del programa, al utilizar el coprocesador, con respecto a la ejecución normal sin su utilización.

El factor de ganancia "G" que se evalúa con el programa simulador, consiste en el porcentaje que supone el número de instrucciones que han sido ejecutadas en su versión "J+1/2", con respecto del número total de instrucciones ejecutadas. Así si de cada 100 instrucciones ejecutadas, "G" lo han sido en su forma "J+1/2" (migrada), el ahorro de tiempo relativo "Rd" (ganancia relativa) obtenido de la misma manera que en la ecuación (23), será :

$$Rd = \frac{100 S - ((100 - G) S + G C)}{100 S} \quad (30)$$

$$Rd = \frac{G}{100} \frac{(S - C)}{S} \quad (31)$$

$$Rd = \frac{G}{100} Rm \quad (32)$$

$$\text{Donde } R_m = \frac{S - C}{S} \quad (33)$$

El significado de "Rm" corresponde a la ganancia relativa off-line, que se obtiene en el caso de que el factor de compactación "F" sea igual a la unidad (en la ecuación (21)), es decir que en primera aproximación :

$$R_d = \frac{G}{100} Re \quad (34)$$

$$G = \frac{R_d}{Re} 100 \% \quad (35)$$

Por lo tanto el valor de "G", en (35), nos da el porcentaje que supone la ganancia relativa que se obtiene con el método presentado, con respecto a la ganancia que se obtendría si el programa hubiera sido totalmente migrado en forma estática, con un factor de compactación unidad.

La forma en que se proporciona la estructura del flujo de control del programa cuya ejecución se quiere simular, se lleva a cabo indicando cuales son los puntos donde supuestamente se encuentran las instrucciones de ruptura de secuencia, ya sean condicionales o incondicionales, señalando del mismo modo el punto destino de la bifurcación. Si la bifurcación es condicional, el tratamiento de ésta se hace mediante dos versiones del simulador, una en forma determinística y otra probabilística.

El tratamiento de las bifurcaciones condicionales en su versión determinística, consiste en proporcionar al simulador, el número de iteraciones que debe realizar el lazo generado por dicha bifurcación, de modo que durante la ejecución simulada del programa, y con el objeto de simular dicho bucle, basta con ir decrementando dicho valor, cada vez que la ejecución alcance dicha instrucción, hasta que la cuenta alcance el cero. Evidentemente este tratamiento sólo permite definir, en el programa cuya ejecución pretendemos simular, flujos de control de ejecución que incluyan segmentos secuenciales, saltos incondicionales y saltos condicionales hacia atrás, estos últimos expresados mediante número de iteraciones del lazo. Sin embargo no es posible incorporar saltos condicionales hacia adelante, dado que en ellos no tiene sentido hablar de número de iteraciones, sino que se producen en general bajo una condición.

Para resolver el problema de los saltos condicionales hacia adelante se ha desarrollado la versión probabilística. Dado que de forma esquemática no podemos pensar en la condición que controla el salto, debido a que su origen puede ser muy diverso, en esta versión se resume el control de la bifurcación en términos de la probabilidad de que se produzca el salto. Para simular la ejecución de dicho salto condicional, cada vez que durante la ejecución del programa se encuentre dicha instrucción, es preciso generar un número aleatorio comprendido entre cero y uno, de manera que permita determinar si se debe realizar el salto o no. Si el

número aleatorio generado es menor o igual que la probabilidad de salto, definida como parámetro de entrada del simulador, implica que debe producirse el salto, en caso contrario se seguirá en secuencia.

Para que este mecanismo probabilístico genere resultados representativos, es necesario que el generador de números aleatorios o pseudoaleatorios utilizado, produzca una secuencia de números, comprendidos entre cero y uno, distribuidos uniformemente y cuya media sea igual a 0.5. Teniendo en cuenta la naturaleza estadística de los resultados, es precisa la ejecución de la simulación un número elevado de veces para dar validez a dichos resultados.

Para facilitar al usuario del simulador, la descripción de los bucles en el programa simulado de forma probabilística, es posible proporcionar los saltos condicionales mediante el número de iteraciones del bucle. Con ello evitamos que en los casos en que dichos saltos generen una estructura iterativa, el usuario deba calcular la probabilidad de salto que tiene la bifurcación que define dicha estructura. La forma de incorporar el número de iteraciones para un bucle en la versión determinista, consiste en relacionar dicho número, con la probabilidad de salto que debe tener la bifurcación condicional al final del bucle.

Supongamos que un programa está constituido por un lazo que termina con un salto condicional al principio, y que dicho salto tiene una probabilidad "P" de producirse, y

evidentemente, una probabilidad "1-P" de seguir en secuencia. Si este programa es ejecutado "M" veces, ¿ cuantas veces en total se habrá producido un salto hacia atrás, en las "M" ejecuciones del programa ?.

El número total de saltos hacia atrás "n" que se producirán vendrá dado por el producto del número de veces que se ha ejecutado el programa, "M", por la probabilidad de que se produzca el salto una sola vez, más "M" por la probabilidad de que se produzca dos veces, ...

$$n = M \cdot (P + P^2 + P^3 + \dots) \quad (36)$$

lo cual corresponde al producto de "M" por la suma de los infinitos términos de una progresión geométrica de razón "P" con "0 < P < 1".

$$n = M \frac{P}{1 - P} \quad (37)$$

Si consideramos el número de ejecuciones "M" del programa suficientemente grande para que los resultados tengan un valor estadístico, el número medio de saltos que se producirá por cada una de las ejecuciones del programa "N" vendrá dado por :

$$N = \frac{P}{1 - P} \quad (38)$$

Si tenemos en cuenta que "N" es el número de saltos hacia atrás que se producen en el lazo, el número de iteraciones "V", será :

$$V = N + 1 \quad (39)$$

A partir de la expresión (38) podemos determinar cual debe ser la probabilidad de salto para la bifurcación terminal de un bucle, para que éste en media y bajo un número grande de ejecuciones, realice un determinado número de iteraciones.

$$P = \frac{V - 1}{V} \quad (40)$$

De este modo en el momento de proporcionar al simulador, un bucle perteneciente al programa simulado, podemos expresarlo bajo número de vueltas e internamente transformarlo en un salto terminal con una cierta probabilidad. Esto permite, como ya se ha dicho, facilitar la entrada del programa simulado para la versión probabilística.

Es claro que para que los resultados obtenidos en el caso probabilístico, la ejecución del programa simulado debe ser repetida un gran número de veces, dado que el tratamiento de los saltos condicionales, en general, supone el manejo de rutinas generadoras de números aleatorios, y por ello los resultados obtenidos son estadísticos.

Se han tomado de forma representativa diferentes programas conteniendo diferentes lazos, donde fuera posible la utilización de las dos versiones del simulador y de esta manera se ha podido comprobar que si en el caso probabilístico, el programa simulado se ejecuta un número de veces superior a 500, los resultados que se obtienen se desvían muy poco de los resultados exactos obtenidos mediante el método determinístico. La consecuencia es que el método probabilístico supone una ejecución del programa simulador más lenta. En contrapartida el método determinístico descrito, con una única ejecución del programa simulado nos proporciona resultados exactos, pero no permite el uso de saltos condicionales hacia adelante.

III.3 PROGRAMA SIMULADOR

En esta sección se describe cada una de las versiones que han sido desarrolladas para la simulación. En primer lugar, vamos a centrar nuestra atención en la simulación determinística. Dentro de esta simulación se han hecho a su vez diferentes versiones, pero todas ellas basadas en un programa central que hemos denominado DETERPRIM y que vamos a estudiar a continuación. Su listado se recoge en el apéndice A.

En la figura III.1a y b se muestra un organigrama general del programa simulador en su primera versión determinista, y que servirá como núcleo para las otras versiones (DETERPRIM).

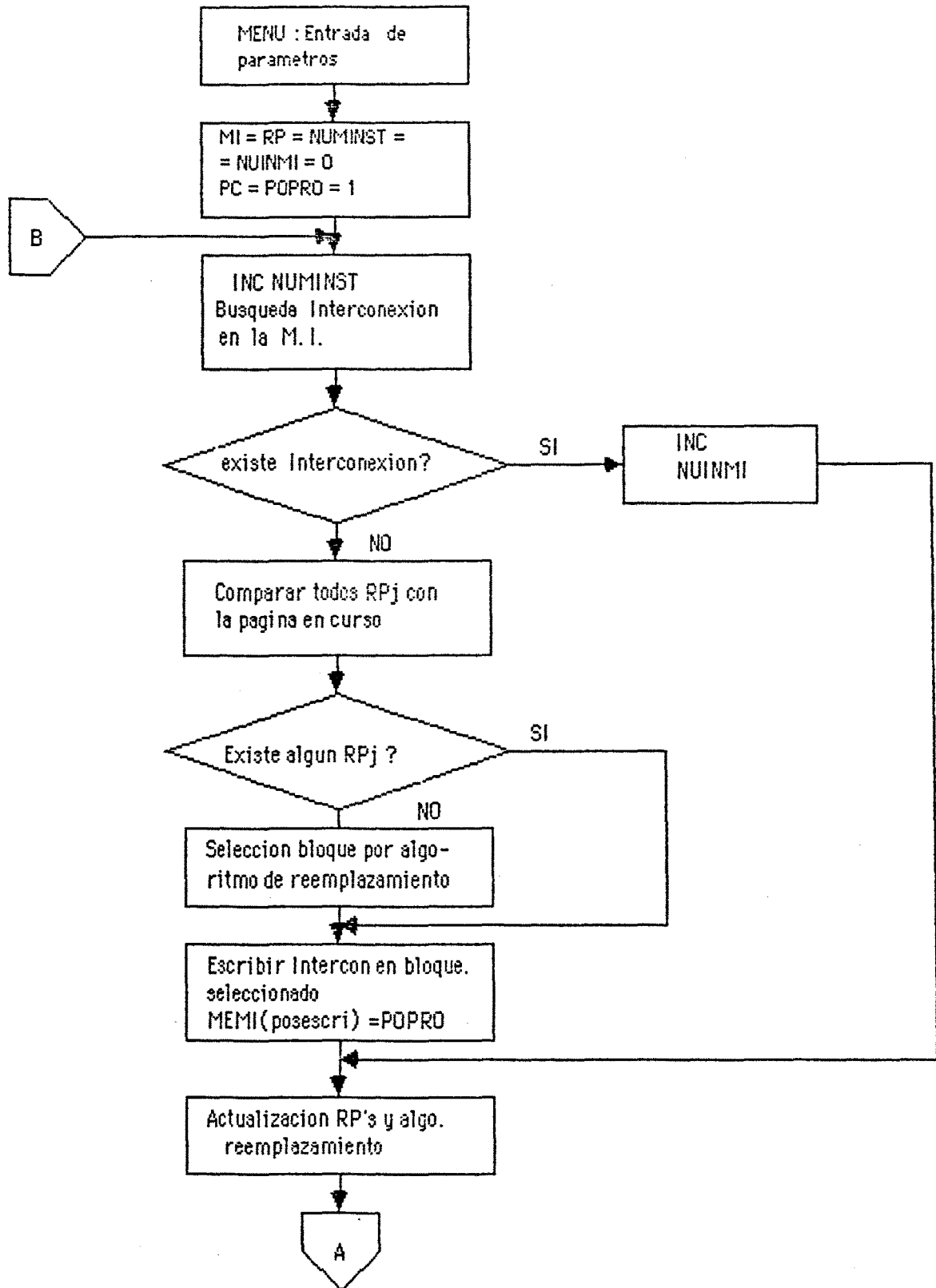


Figura III.1.a Simulador DETERPRIM

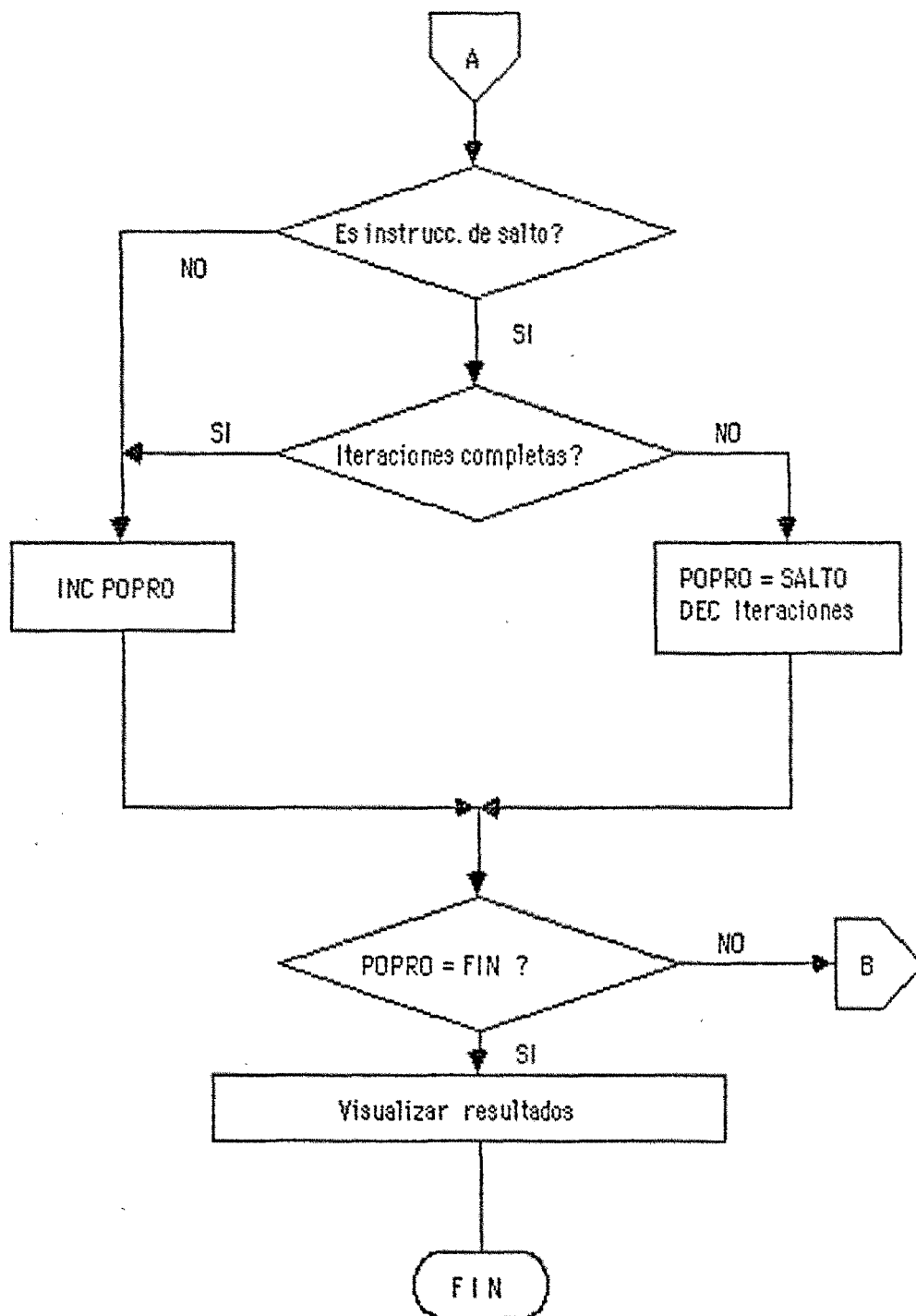


Figura III.1.b Simulador DETERPRIM

3.1 Entrada de parámetros

En él podemos ver en un primer bloque, toda la parte correspondiente a la entrada de parámetros que el usuario debe proporcionar al simulador de forma interactiva. Esto se realiza mediante menú, en el que aparecen diferentes opciones :

Una opción "4" (RESET) que permite inicializar el simulador y proporcionarle toda la información que precisa. Esta información se configura proporcionando :

La longitud del programa, dada como un número decimal. El programa a simular empieza siempre en la posición "1". Estamos suponiendo que el programa pertenece a un sistema cuyo repertorio es de una sola palabra por instrucción, puesto que es el caso más desfavorable que se presenta para nuestro método. Procedimiento LONGLISTA. El programa simulado es creado como un array de registros (records), cada uno de los cuales está formado por tres campos de enteros que sólo tienen significación en caso de una instrucción de ruptura de secuencia, en caso contrario el contenido de estos tres campos deberá ser cero : "SALTO" cuyo contenido es la dirección de salto. "VUELTAS" cuyo contenido indica el número de saltos que debe realizar esta instrucción para completar un bucle. CUVU cuyo contenido inicialmente es igual al de VUELTAS, y va decrementándose a

medida que se completen las iteraciones del bucle. Terminada la ejecución de un lazo, en todas sus iteraciones, será necesario reinicializar el contenido de CUVU con el de VUELTAS para posteriores ejecuciones de dicho lazo, por ejemplo en las anidaciones.

El número de bloques en que se desea particionar la M.I. y el tamaño de cada uno de ellos. Procedimiento MEMINTER. La M.I., en el simulador, es vista como un vector (array) de enteros donde cada una de sus componentes constituye una posición de M.I.. La división de M.I. en bloques sólo es necesaria en el momento de acceder a la M.I. para la búsqueda o creación de una interconexión. La creación de una interconexión en una de estas componentes (posiciones), implica a nivel del simulador, almacenar el valor del contador de programa en la posición de M.I., cuya dirección es calculada a través de la posición de la instrucción en la página, el tamaño de ésta y el número de página (dirección de página).

Definir el número de registros de página (historia) que se desee que tenga cada uno de los bloques (formando un LIFO de registros). Procedimiento REGISTROS. Normalmente el número utilizado será de "uno" pues como ya se ha dicho con anterioridad, aumentar su número no supone una mejora del comportamiento del sistema.

Describir de forma esquemática el flujo de control del programa (procedimiento ENTRARVAL). El simulador nos preguntará el origen y el destino de los saltos (para formar bucles), así como el número de veces "V" que se deba saltar hacia atrás para ejecutar cada bucle. El número de iteraciones del bucle será "V+1". En caso de pretender un salto incondicional a alguna posición del programa simulado, basta con indicar que el número de saltos sea "32767", que corresponde al número entero estandar más grande que se permite en UCSD Pascal versión APPLE II. Toda instrucción que no sea especificada, el simulador supone que es seguida por su consecutiva en el programa simulado.

Por último dar la selección del algoritmo a utilizar durante la simulación del programa (procedimiento SELAL). Inicialmente existen tres opciones : Mapping Directo, Distancia Máxima y L.R.U.. El algoritmo de Mapping Directo no ha sido mencionado antes por su bajo rendimiento y poca adaptabilidad, por lo que tampoco ha sido utilizado con profusión. Este algoritmo de reemplazamiento consiste en asignar los bloques de M.I. consecutivos a páginas consecutivas de la memoria principal. Así si disponemos de "B" bloques, el primero será asignado a las páginas : 1, B+1, 2B+1, El segundo a las páginas 2, B+2, 2B+2,

Las otras opciones que se ofrecen en el menú,

permiten la visualización e introducción selectiva de los parámetros mencionados :

Una opción "0" que permite la visualización de todos los parámetros seleccionados, y que son los que el simulador tomará por defecto; algoritmo, número de registros de historia, número de bloques de M.I. y su tamaño, y la descripción del programa simulado. El formato de salida por pantalla, a modo de ejemplo sería :

```
DETERPRIM      ALGORITMO DISTANCIA
HISTORIA = 1   BLOQUES = 4 DE 2
PROGRAMA DE 10 CON LOS LAZOS :
  1  <----- 10 ( 9)
  3  <-----  5 (14)
```

En este ejemplo (DETERPRIM es el nombre de la versión del programa simulador) se ha seleccionado el algoritmo de Distancia Máxima, cuatro bloques de M.I. de dos posiciones cada uno y el programa simulado consta de diez instrucciones y está constituido por dos bucles anidados, uno exterior (1<---10) que se ejecuta diez veces (9+1) y uno interior (3<---5) que se ejecuta quince veces (14+1).

Una opción "1" destinada a cambiar el número de bloques en que se particiona la M.I. y su tamaño, permaneciendo inalterable todo lo demás.

Una opción "2" que permite cambiar el número de

registros de página (historia) asignados a cada bloque.

Una opción "3" para cambiar la estructura del programa simulado, a nivel de los bucles y saltos que lo conforman.

Una opción "5" selecciona el algoritmo de reemplazamiento entre los tres antes mencionados.

Una opción "6" genera la ejecución de la simulación propiamente dicha.

Una opción "7" permite la salida de este menú principal y del programa simulador.

3.2 Inicialización

En un segundo bloque del organigrama, se inicia propiamente la ejecución del simulador, en él se incluyen todas las funciones de inicialización de las variables del programa simulador, como son :

a- Los contenidos de todas las posiciones de la M.I. simulada MEMI(A) deben ser inicializados a cero mediante el procedimiento INIMEMI para que en su utilización pueda determinarse si se encuentran vacías.

b- Los contenidos correspondientes a los registros de

página R.P. y que en el programa son denominados "registros de historia", son inicializados cada uno con un valor negativo diferente, para que de este modo, cuando todos los bloques se encuentran vacíos, pueda actuar el algoritmo de distancia máxima, si éste fuera el seleccionado. Procedimiento INIREG.

c- En caso de que el algoritmo que se seleccione sea el L.R.U. deberemos también inicializar los registros que lo integran, de modo que cada uno de ellos señale un bloque distinto. Procedimiento INILRU.

d- Es necesaria la utilización de un contador de programa que señale la instrucción en ejecución simulada, este contador de programa viene representado por la variable "POPRO" (posición en el programa). Esta variable debe ser inicializada a uno al comienzo de la ejecución del programa simulado, para señalar la primera instrucción a ejecutar. También es necesario inicializar a cero la variable que llevará la cuenta de las instrucciones que se ejecutan en total "NUMINST", y las que se ejecutan en su forma migrada, es decir, las que son encontradas en alguna posición de M.I. "NUINMI".

3.3 Ejecución simulada

Una vez terminada la inicialización de variables, se entra en un lazo hasta que el contenido del contador de

programa (POPRO) sea igual a la longitud que se ha asignado al programa.

Supongamos que se inicia la ejecución de la primera instrucción del programa simulado, por ello incrementamos en uno la cuenta total de instrucciones ejecutadas (NUMINST) en el programa simulado. Iniciamos la búsqueda, en todos los bloques de M.I., de la interconexión de la instrucción en curso mediante el procedimiento BUSQUEDA. En este procedimiento, en primer lugar debemos calcular la dirección correspondiente a la posición que se debe leer en todos los bloques, para determinar la existencia de la interconexión. Se efectúa la lectura de esta posición en todos los bloques de M.I. y mediante la variable booleana "ESTA" se indica la presencia o no de la interconexión.

Caso de que la interconexión se encuentre en algún bloque de M.I., se incrementa la cuenta de "NUINMI", instrucciones ejecutadas en versión "J+1/2" y se inicia la búsqueda de la siguiente instrucción.

3.4 Fallo de página

Caso de que la interconexión no se encuentre en ningún bloque de M.I., es preciso seleccionar un bloque para realizar en él la escritura de la interconexión de nueva creación, para la instrucción en curso. En primer lugar el procedimiento HISTORIA analiza si existe algún bloque cuyo registro de página haga referencia a la página de la memoria

principal a la que pertenece la instrucción en ejecución. Si es así, se indica mediante la variable booleana "EXBO" (existe bloque) y el bloque encontrado es referenciado mediante la variable "BLOQUE".

3.5 Algoritmo de reemplazamiento

Si no se encuentra ningún bloque que cumpla esta condición, es necesario recurrir al algoritmo de reemplazamiento para que seleccione un bloque donde realizar la escritura. En función del contenido de la variable "ALGO", podemos determinar cual es el algoritmo seleccionado por el usuario, y según sea dicha elección se ejecutan alternativamente los procedimientos "MAPDIRECT", "DISTANCIA" o "LRU", los cuales realizarán la selección e indicarán el bloque elegido en la variable "BLOQUE". Los procedimientos correspondientes al Mapping Directo y al L.R.U. son muy simples y de fácil comprensión en el listado que se proporciona en su apéndice. El procedimiento correspondiente al de Distancia Máxima viene ilustrado en el organigrama en la figura III.2a y b.

En la variable "C" iremos guardando temporalmente, la mayor distancia que se encuentre en las sucesivas comparaciones entre la página de la memoria principal en curso y el contenido del R.P. de cada bloque. La variable "B" señala el bloque que se está comparando en cada momento.

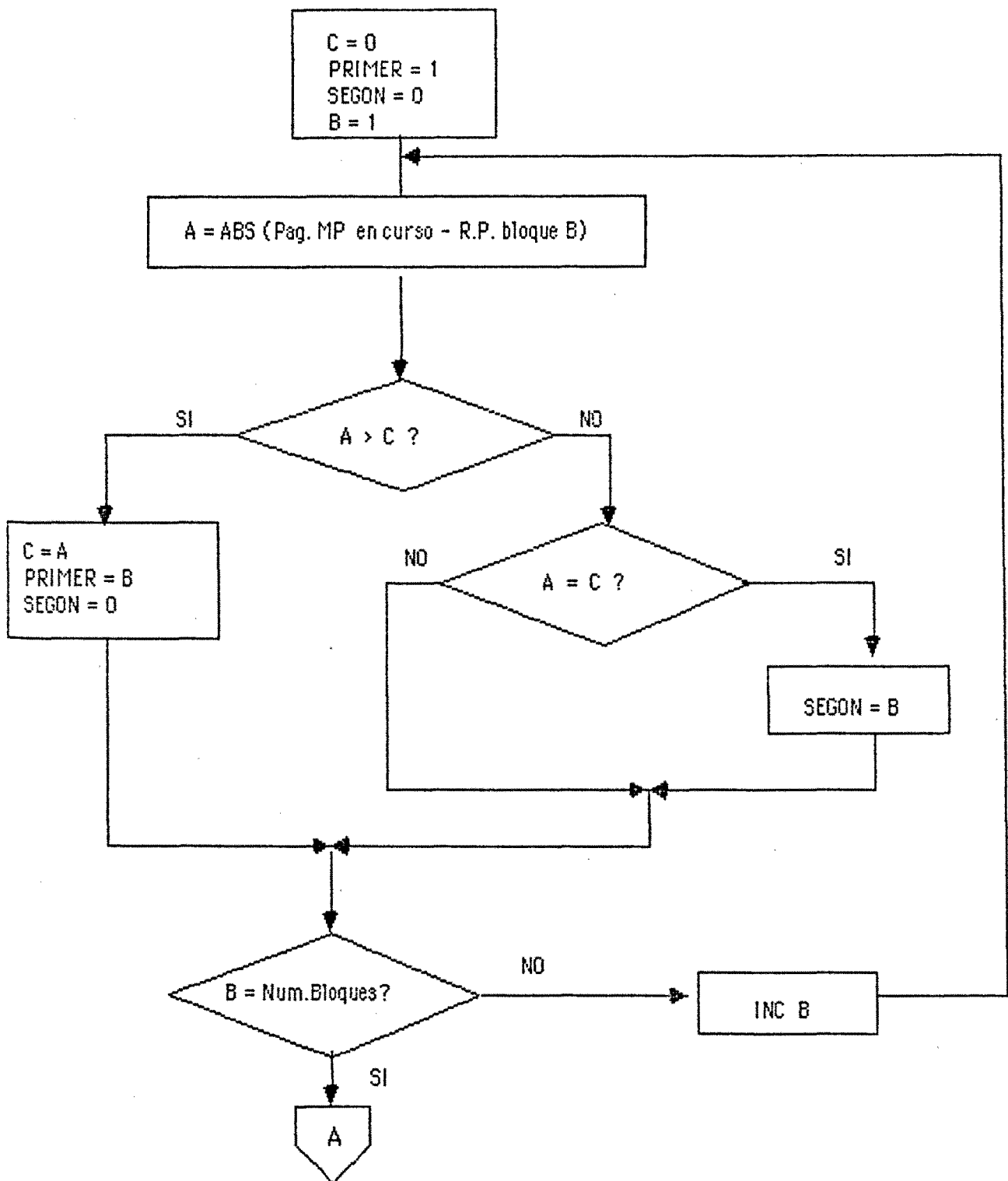


Figura III.2.a Procedimiento Algoritmo Distancia

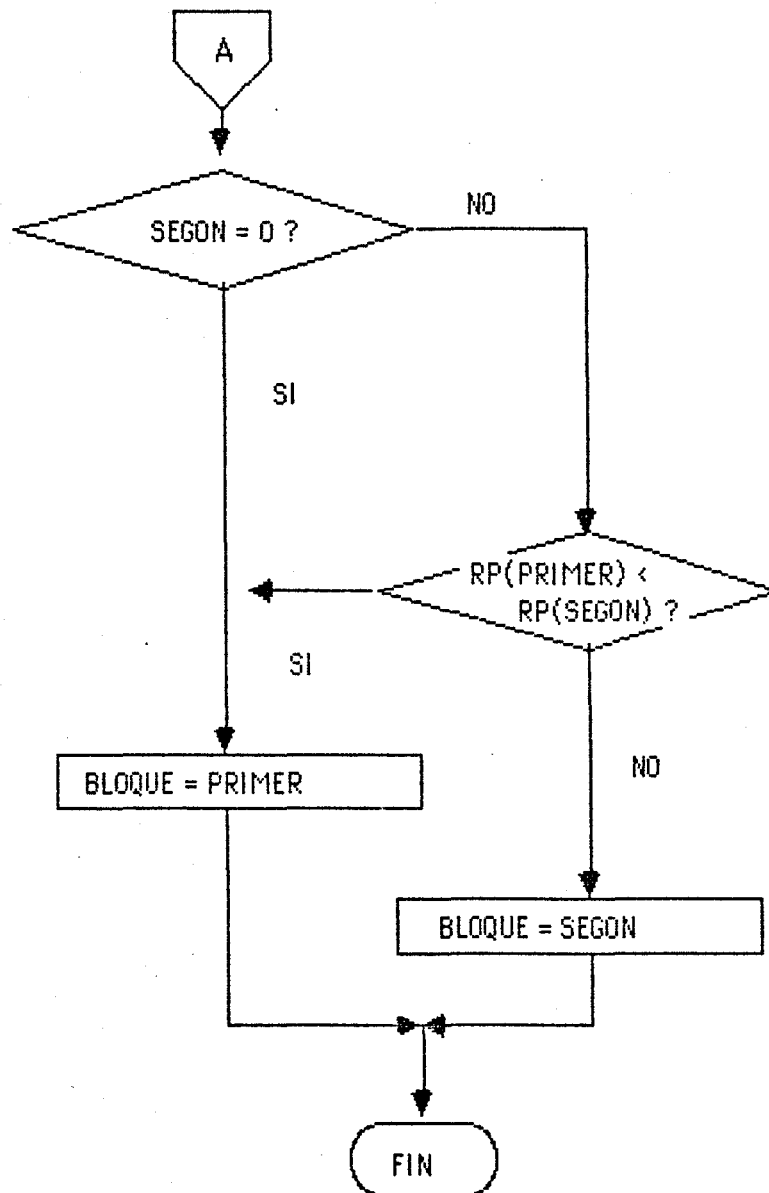


Figura III.2.b Procedimiento Algoritmo Distancia

Puede ocurrir que dos bloques contengan páginas equidistantes de la página en curso, resolvemos el conflicto eligiendo de entre las dos, aquella cuya dirección de página sea menor. Para ello son necesarias dos nuevas variables "PRIMER" y "SEGON". Si la diferencia del contenido del R.P. con la página (A), es la más grande encontrada hasta el momento (C), se actualiza "C" y "PRIMER" pasa a señalar a dicho bloque. Si no fuera así hay que ver si hubo algún bloque, comparado anteriormente, que estuviera tan lejos de la página como éste. Si esto ocurre, este último estará en "PRIMER" y el actual se señala por "SEGON". Una vez analizados los R.P. de todos los bloques, debemos comprobar si ha ocurrido la situación que acabamos de describir, preguntando por "SEGON". Si su contenido es cero, no se ha producido equidistancia y "PRIMER" señala el bloque más lejano y por tanto el seleccionado. Si su contenido es diferente de cero, significa que existe equidistancia de dos bloques con la página de M.P. en curso y hay que resolverla viendo cual de los R.P. de los dos bloques contiene un menor contenido. Este será el bloque seleccionado.

3.6 Creación de la interconexión

Volviendo al organigrama general de DETERPRIM, y conocido el bloque donde se debe realizar la escritura de la nueva interconexión, el siguiente paso consiste en la escritura propiamente dicha, almacenando el contenido actual de "POPRO" en la posición de M.I. determinada por "POSESCRI". Dado que en el simulador la M.I. es vista como un continuo,

la variable "POSESCRI" indica la dirección absoluta de la posición a escribir, la cual se calcula a partir de la posición dentro del bloque, el número del bloque seleccionado y su tamaño.

3.7 Actualización del algoritmo

Terminada la escritura de la interconexión, se procede a actualizar los registros de página mediante el procedimiento "ACTUALHIS", en el cual el R.F. del bloque donde se ha hecho la escritura es cargado con el valor de la nueva página a la que hace referencia el bloque.

En el caso de que el algoritmo seleccionado sea el L.R.U. y la M.I. haya sido particionada en más de un bloque, es necesaria la actualización de dicho algoritmo (Procedimiento ACTUALRU), tanto en el caso de que la interconexión ya existiera en M.I., como si no. En este procedimiento, independientemente del registro ("LUGAR") que ocupaba el bloque que contenía la interconexión, o el bloque en el que se ha escrito la misma (LUGAR), pasa a ocupar el registro cabecera del algoritmo, y todos los que estuvieran por encima del registro ("LUGAR") que ocupaba éste, son empujados hacia abajo, quedando en el fondo el que será seleccionado en la siguiente utilización del algoritmo.

En este punto termina el tratamiento de las interconexiones, y la simulación del funcionamiento del coproce-

sador. Notemos que lo dicho hasta ahora se ajusta totalmente a la descripción que se ha hecho, al final de capítulo I, del funcionamiento del sistema.

3.7 El secuenciamiento

Iniciamos ahora la búsqueda de la siguiente instrucción a ejecutar.

Si el campo de salto de la instrucción es cero, es decir que no se trata de una instrucción de bifurcación; o el campo que cuenta las iteraciones hechas en el bucle (CUVU) ha llegado a cero, debemos reinicializar dicho contador con su valor inicial (campo de VUELTAS) (para posibles posteriores ejecuciones del bucle en anidaciones), y seguir en secuencia incrementando el contador de programa (POPRO) en uno. Si se trata de una bifurcación en un bucle, cuyas iteraciones no se han completado, se decrementa el contador de iteraciones (CUVU), y se actualiza POPRO con el contenido del campo de "SALTO". Una vez POPRO haya alcanzado un valor igual a la longitud del programa elegida por el usuario, cesará la ejecución del programa simulado, calculándose la ganancia obtenida en la simulación :

$$G = \frac{\text{NUINMI}}{\text{NUMINST}} \quad 100 \% \quad (41)$$

3.8 Visualización de resultados

El último paso corresponde a la visualización y en

su caso impresión de los resultados obtenidos (Procedimiento RESULTADOS), regresando de nuevo al menú principal, donde se podrán realizar los cambios que se deseen en los parámetros y ejecutar de nuevo bajo la nueva configuración.

Para mostrar el formato final en que se obtiene la información, veamos cual sería el resultado de la ejecución simulada del ejemplo propuesto en la opción "0" del menú principal :

```
DETERPRIM      ALGORITMO DISTANCIA
NORMAL = 520   M. I. = 474   GANAN = 9.11538E1 %
HISTORIA = 1   BLOQUES = 4   DE 2
PROGRAMA DE 10 CON LOS LAZOS :
  1  <----- 10 ( 9)
  3  <-----  5 ( 14)
```

En el formato en que se presenta el resultado de la ejecución simulada, podemos ver que además de la información referente a los parámetros del simulador, que ya se presentaban en la opción "0" del menú principal, en negrita aparece el número total de instrucciones ejecutadas en el programa, el número de instrucciones ejecutadas en su versión "J+1/2", es decir aquellas cuya interconexión ha sido encontrada durante su ejecución (M.I. = 474) y la ganancia (GANAN) obtenida, es decir, el porcentaje de instrucciones ejecutadas en versión "J+1/2" con respecto del total de las ejecutadas.

3.9 Simulación sistemática

Existe una versión modificada de este programa determinístico (DETERPRIM) y que hemos llamado "DETSISMI", el cual en su núcleo central de operación es idéntico al anterior, permitiendo además que dado un tamaño total de M.I., ejecutar el programa simulado sistemáticamente, para todas las posibles particiones que se puedan hacer de la M.I.. Básicamente la estructura de este programa podemos resumirla así :

```
PROGRAM DETSISMI;
BEGIN
  FOR NUM. DE BLOQUES := 1 TO TAMAÑO TOTAL DE MI DO
    BEGIN
      IF TAM. TOTAL MI MOD NUM. BLOQUES = 0 THEN
        BEGIN
          TAMAÑO BLOQUE := TAM. TOTAL MI DIV NUM. BLOQUES;
          DETERPRIM;
        END;
      END;
    END.
END.
```

El listado correspondiente a esta versión del programa simulador (DETSISMI) se muestra en el apéndice B.

3.10 Simulación probabilística

La versión probabilística del programa simulador ha sido desarrollada tomando como base la primera versión del determinístico (DETERPRIM), modificando solamente aquellas partes del programa que hacen referencia al secuenciamiento de las instrucciones del programa simulado, que es el aspecto que lo diferencia del mismo.

La finalidad de la versión probabilística del simulador, como ya se ha dicho repetidas veces, pretende la utilización de las bifurcaciones condicionales en el programa simulado, con objeto de poder simular cualquier estructura de programación, por compleja que sea.

Esta versión probabilista, recibe el nombre de "BRANCLRU" y se incluye su listado en el apéndice C.

Para realizar la simulación con esta versión, es necesario ejecutar el programa simulado un gran número de veces. Este número de ejecuciones deberá ser elegido por el usuario en el menú principal de esta versión, y normalmente oscilará entre 500 y 1000 para obtener resultados que puedan ser considerados como fiables. La variable del simulador que lleva la cuenta de dichas ejecuciones es "VECES". Del mismo modo, en esta versión, se precisa la utilización de las variables : NUSU y NUMISU, las cuales irán acumulando el número total de instrucciones ejecutadas en el programa simulado y las ejecutadas en versión "J+1/2" respectiva-

mente, a medida que se produzcan las sucesivas ejecuciones del programa simulado. Así, el número medio de instrucciones ejecutadas en el programa simulado, "NUTO", será :

$$NUTO = \frac{NUSU}{VECES} \quad (42)$$

El número medio de instrucciones ejecutadas en versión "J+1/2", "NUMITO" vendrá expresado como :

$$NUMITO = \frac{NUMISU}{VECES} \quad (43)$$

El programa simulador "BRANCLRU", en forma esquemática quedará descrito como :

```
PROGRAM BRANCLRU;
BEGIN
  NUSU := 0;
  NUMISU := 0;
  FOR X := 1 TO VECES DO
    BEGIN
      INIMEMI;
      INIREG;
      EJECUCION
      NUSU := NUSU + NUMINST;
      NUMISU := NUMISU + NUINMI;
    END;
END;
```

```

NUTO := NUSU / VECES;
NUMITO := NUMISU / VECES;
GANAN := ( NUMITO / NUTO ) * 100;

RESULTADOS;

END.

```

En este programa simulador se ha creado un procedimiento, "EJECUCION" el cual está constituido por la parte operativa de la primera versión determinista (DETERPRIM), modificando el secuenciamiento de las instrucciones, como ya se ha dicho. Al igual que en DETERPRIM, el programa simulado es creado como un array de registros (records), cada uno de los cuales ahora solamente está formado por dos campos, uno entero y otro real, que sólo tienen significación en caso de una instrucción de ruptura de secuencia, en caso contrario el contenido de estos dos campos deberá ser cero: "SALTO" (entero) cuyo contenido es la dirección de salto y "PRO" (real) cuyo contenido indica la probabilidad de salto que tiene la instrucción de bifurcación condicional. Ambos campos son inicializados por el usuario con las opciones del menú principal que permiten la definición del programa simulado. recordemos que en el caso de haber expresado algún lazo del programa como número de vueltas, estas son traducidas salto con cierta probabilidad, en la bifurcación terminal del bucle.

En la figura III.a y b se muestra el organigrama correspondiente al procedimiento EJECUCION. en él podemos ver que en cuanto a la parte operativa del manejo de las

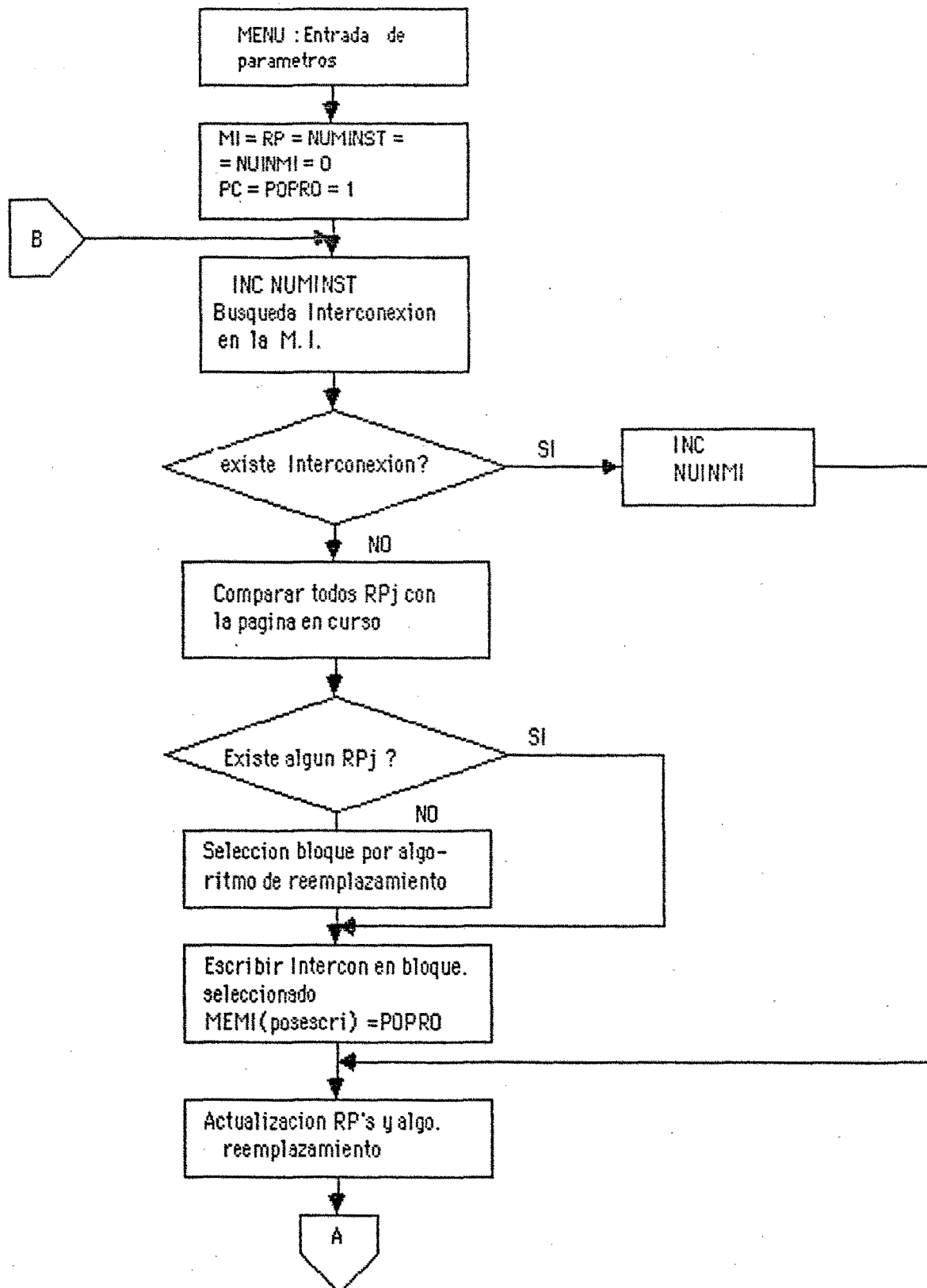


Figura III.3.a Procedimiento EJECCION (BRANCLRU)

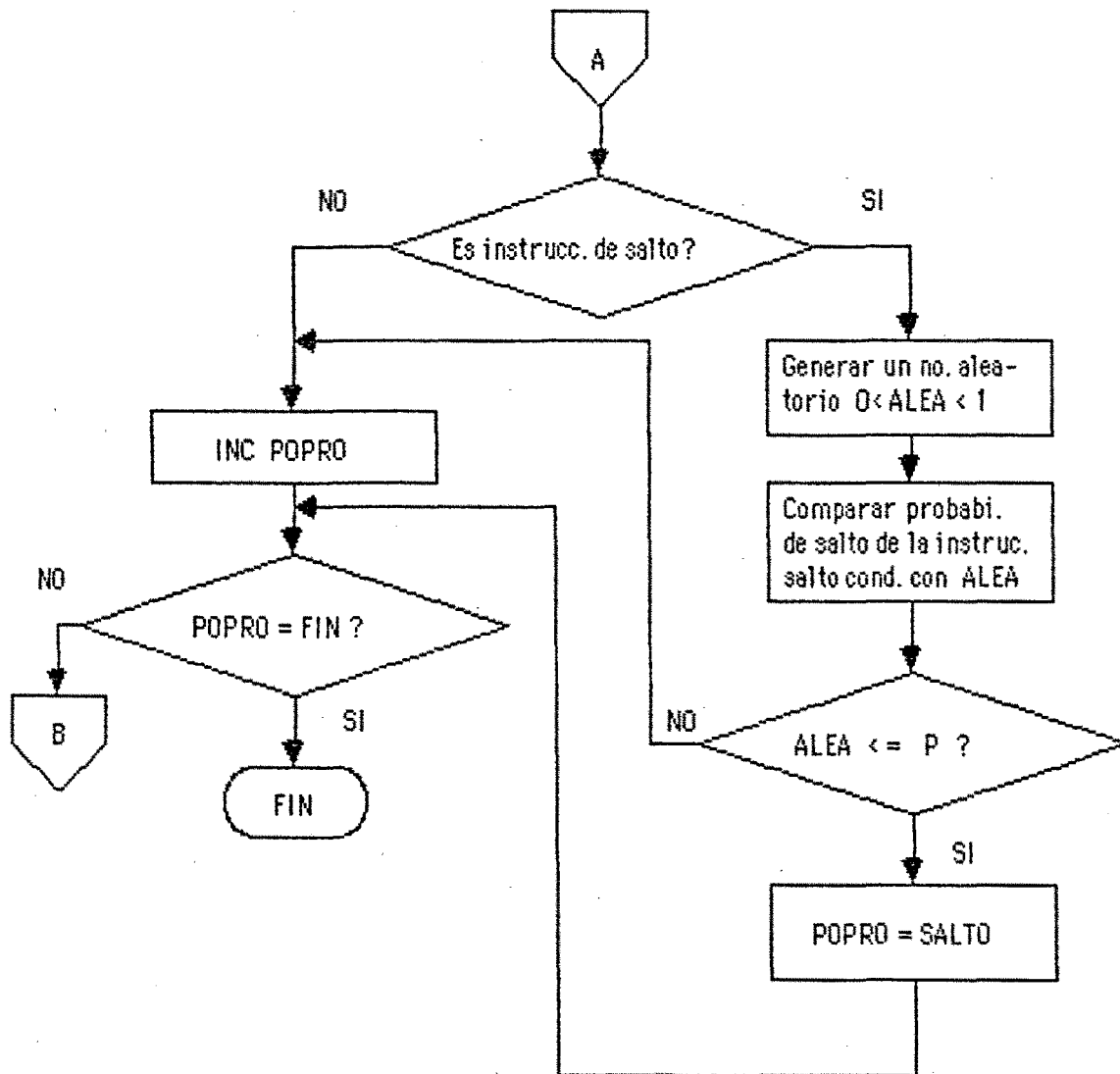


Figura III.3.b Procedimiento EJECUCION (BRANCLRU)

interconexiones, con respecto a su búsqueda y escritura en la M.I. hasta la actualización del algoritmo de reemplazamiento, es idéntica a la correspondiente al programa "DETER-PRIM". En cuanto al secuenciamiento de las instrucciones, en caso de no tratarse de una instrucción de bifurcación condicional, se incrementa el contador de programa (POPRO) y se ejecuta la instrucción siguiente. En el caso de tratarse de una bifurcación, para evaluar si la condición se cumple o no, es decir, si se debe realizar el salto, es necesario generar un número aleatorio comprendido entre cero y la unidad. La rutina que se ha usado para la generación de números pseudoaleatorios corresponde a la perteneciente a la librería de rutinas PASCAL del sistema. Esta rutina produce una secuencia cuya media se sitúa en 0.5 y la probabilidad de aparición de los diversos números aleatorios es igual para todos ellos (distribución uniforme).

Obtenido el número aleatorio (ALEA), se debe comparar éste con la probabilidad de salto de la instrucción, es decir con el campo "PRO" de la instrucción. Si el número aleatorio obtenido es menor o igual que la probabilidad de salto asignada a la instrucción, significa que la condición de bifurcación se cumple y el salto deberá llevarse a cabo. Para ello deberá actualizarse el contador de programa "POPRO" con el campo de "SALTO" de la instrucción. En caso de que la condición no se cumpla, es decir que el número aleatorio se mayor que "PRO", la ejecución del programa simulado deberá seguir en secuencia y por ello incrementado

el contador de programa en uno.

Debido a la necesidad de realizar la simulación de programas con un gran número de datos de entrada, tanto en el caso determinístico como en el probabilístico y especialmente en este último, se han creado versiones de ambos para ser ejecutados en un sistema VAX-11/785, puesto que el microordenador APPLE II plus sobre el que inicialmente se ejecutaba dicho simulador, precisaba de tiempos de cálculo que en algunos casos superaba las 72 horas, con los consiguientes problemas que ello comporta. Estas nuevas versiones no se diferencian substancialmente de las descritas en esta memoria, y por ello no son detalladas en la misma.

III.4 INCIDENCIA DE LOS ALGORITMOS DE REEMPLAZAMIENTO EN EL COMPORTAMIENTO DEL SISTEMA

Una primera utilización del programa simulador ha consistido en la valoración de la incidencia de los diferentes tipos de algoritmos de reemplazamiento así como del tamaño de la memoria de interconexión y los efectos de la partición sobre el comportamiento del sistema en cuanto a las ganancias obtenidas.

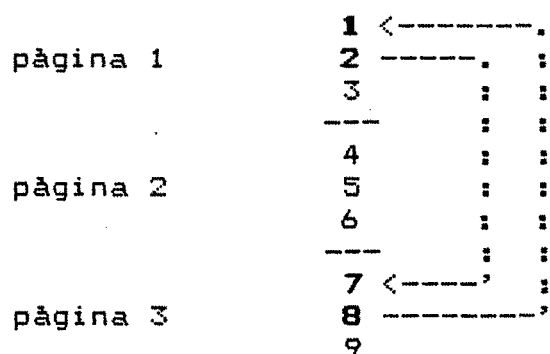
4.1 Algoritmo de Mapping Directo

El primer caso estudiado corresponde al del algoritmo de Mapping Directo. Con él se ha visto que los resultados obtenidos producían ganancias de muy bajo orden, debi-

do principalmente al hecho de que se dan casos en los cuales a pesar de que un lazo, por su tamaño, quepa en la M.I., la ganancia que se obtiene es nula. El motivo de tal resultado podemos visualizarlo con el siguiente ejemplo :

Sea la configuración formada por una M.I. de 2 bloques de 3 posiciones cada uno. Las páginas en que virtualmente vemos dividida la memoria principal tendrán también tres palabras cada una. El paginado de la memoria principal y el programa simulado vienen descritos por :

EJEMPLO III.1



es decir, se trata de un lazo de cuatro instrucciones (menor que el tamaño total de M.I.) dividido en dos segmentos distribuidos en dos páginas no consecutivas, ocupando las mismas posiciones en ambas páginas. Durante la ejecución del programa, los bloques irán siendo completados de la siguiente forma :

ciclos ejec.	T	T+1	T+2	T+3	T+4		
BLOQUE A	1	1	7	7	1	posición	1
(página 1)	x	2	2	8	8	"	2
	x	x	x	x	x	"	3
BLOQUE B	x	x	x	x	x	"	1
(página 2)	x	x	x	x	x	"	2
	x	x	x	x	x	"	3

De este modo vemos que al ir ejecutándose las instrucciones, las interconexiones que se van generando son escritas encima de las que serán necesarias a posteriori, produciéndose una destrucción de información innecesaria por el tamaño de M.I. disponible. Los resultados obtenidos utilizando el simulador "DETERPRIM" son :

```

DETERPRIM    MAPPING DIRECTO
NORMAL = 25  M.I. = 0  GANAN = 0.00000 %
HISTORIA = 1  BLOQUES = 2 DE 3
PROGRAMA DE 9 CON LOS LAZOS  :
    7 <---- 2 (32767)
    1 <---- 8 ( 5)

```

Por tanto podemos concluir que con el algoritmo de Mapping Directo el comportamiento del sistema no sólo depende de la estructura del programa sino que además depende fuertemente de la posición que éste ocupe en la memoria principal. En definitiva el comportamiento del sistema se reduce a la situación en que la M.I. estuviera formada por un único bloque, sin partición alguna.

4.2 Algoritmo Last Recently Used

El segundo algoritmo de reemplazamiento que se ha estudiado ha sido el L.R.U. (menos recientemente utilizado) y con él se eliminan problemas como el que se acaba de describir, garantizando que si un lazo, por su tamaño total, cabe en la M.I., no producirá nunca una ganancia nula. El ejemplo III.1, desarrollado para el caso de Mapping Directo, en el caso de L.R.U. arroja los siguientes resultados :

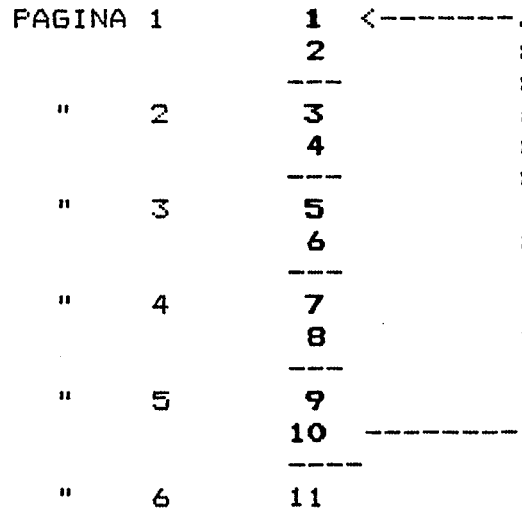
EJEMPLO III.2

```
DETERPRIM   L. R. U.
NORMAL = 25  M.I. = 20  GANAN = 8.00000E1 %
HISTORIA = 1  BLOQUES = 2 DE 3
PROGRAMA DE 9 CON LOS LAZOS  :
    7 <---- 2 (32767)
    1 <---- 8 (    5)
```

El contraejemplo para este algoritmo (Ba80) consiste en que para aquellos lazos cuyo tamaño sea una página mayor que el tamaño de M.I., la ganancia se reduce a cero. Esto es debido a que en una situación así, el L.R.U. hace que el bloque seleccionado cada vez, sea justamente el que se va a necesitar inmediatamente en la ejecución. Veámoslo con un ejemplo.

Supongamos una M.I. particionada en cuatro bloques de dos posiciones cada uno, y un lazo como el que se describe :

EJEMPLO III.3



Durante la ejecución de la primera vuelta del lazo, el contenido de los bloques quedará:

	BLOQUE A (pag 1)	BLOQUE B (pag 2)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	1	3	5	7
p2	2	4	6	8

a partir del momento en que todos los bloques de M.I. están llenos empieza el reemplazamiento, ahora es necesario ubicar la página 5 del programa. La menos recientemente utilizada es la "1", justo la que nos haría falta encontrar una vez se haya ejecutado la "5".

	BLOQUE A (pag 5)	BLOQUE B (pag 2)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	9	3	5	7
p2	10	4	6	8

Una vez ejecutada la página "5", hay que ubicar de nuevo la "1" y ésta será colocada en el bloque que almacena las interconexiones de la página "2", que a su vez es la menos recientemente utilizada pero próximamente necesaria :

	BLOQUE A (pag 5)	BLOQUE B (pag 1)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	9	1	5	7
p2	10	2	6	8

Como este proceso se repetirá durante todas las iteraciones, la ganancia que se obtendrá en este caso es exactamente cero. Los resultados del simulador confirman lo expuesto :

```

DETERPRIM          L. R. U.
NORMAL = 101 M.I. = 0  GANAN = 0.00000 %
HISTOR = 1  BLOQUES = 4 DE 2
PROGRAMA DE 11 CON LOS LAZOS :
    1 <---- 10 ( 9)

```

Sin embargo el comportamiento del algoritmo de Mapping Directo para este caso es mucho mejor :

DETERPRIM **MAPPING DIRECTO**

NORMAL = 101 M.I. = 54 GANAN = 5.34653E1 %

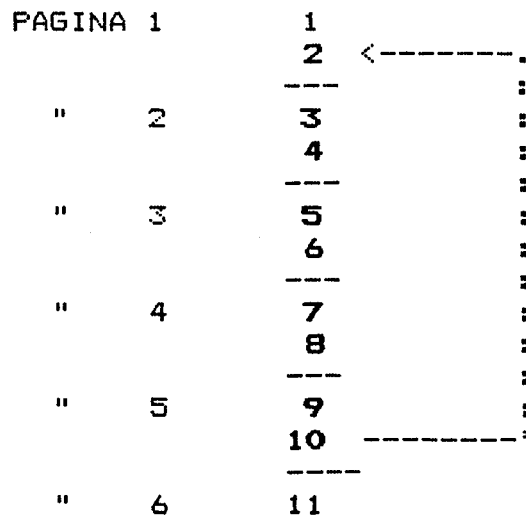
HISTOR = 1 BLOQUES = 4 DE 2

PROGRAMA DE 11 CON LOS LAZOS :

1 <---- 10 (9)

Si observamos el siguiente ejemplo, para L.R.U., con la misma partici3n de M.I. pero reduciendo el tama1o del lazo a una situaci3n intermedia entre que el el tama1o del lazo sea menor o igual al de M.I. y que supere a 3ste en una p3gina :

EJEMPLO III.4



	BLOQUE A (pag 1)	BLOQUE B (pag 2)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	1	3	5	7
p2	2	4	6	8

	BLOQUE A (pag 5)	BLOQUE B (pag 2)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	9	3	5	7
p2	10	4	6	8

	BLOQUE A (pag 5)	BLOQUE B (pag 1)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	9	3	5	7
p2	10	2	6	8

	BLOQUE A (pag 5)	BLOQUE B (pag 2)	BLOQUE C (pag 3)	BLOQUE D (pag 4)
p1	9	3	5	7
p2	10	4	6	8

Podemos ver que se sigue ganando a pesar de que el lazo sea mayor que el tamaño de M.I.. Esto es debido a que el borrado en los bloques reasignados no es total sino selectivo, lo que hace que los residuos que se producen por ello, beneficien enormemente al comportamiento del sistema. Todos los valores en negrita corresponden a instrucciones cuyas interconexiones son encontradas en M.I.. Los resultados del simulador :

```

DETERPRIM          L. R. U.
NORMAL = 92  M.I. = 36  GANAN = 3.91304E1 %
HISTOR = 1  BLOQUES = 4 DE 2
PROGRAMA DE 11 CON LOS LAZOS :
      2 <---- 10 ( 9)

```

En este caso el algoritmo de Mapping Directo, a pesar de sus graves defectos, es también superior al del L.R.U. :

```
DETERPRIM          MAPPING DIRECTO
NORMAL = 92  M.I. = 63  GANAN = 6.847831E1 %
HISTOR = 1  BLOQUES = 4 DE 2
PROGRAMA DE 11 CON LOS LAZOS :
    2 <---- 10 ( 9)
```

Si el borrado de los contenidos de las posiciones de los bloques reasignados no hubiera sido selectivo, sino total, como ocurre en los esquemas de memoria virtual o en los de memoria cache, en este ejemplo la ganancia hubiera sido cero.

4.3 Efecto de la partición de la M.I. en el comportamiento

Veamos cual es el efecto que produce la partición de la M.I. en bloques sobre el comportamiento del sistema. Si consideramos la M.I. constituida por un único bloque, evidentemente no es necesario el uso de registros de página ni de algoritmos de reemplazamiento, lo cual simplificaría la circuitería a utilizar y como consecuencia el coste. El efecto negativo que se obtiene es la falta de adaptación de la M.I. al perfil de ejecución del programa. Para lazos formados por un único segmento de posiciones consecutivas,

ciones tenga un formato de más de una palabra, si la partición no es la máxima, normalmente entre dos interconexiones existirán huecos. Esto se evita en el caso de partición máxima donde no existirán huecos entre interconexiones. Esto tampoco supone una enorme ventaja puesto que si quedan huecos no implica necesariamente que se desaproveche la M.I., pues existe una probabilidad de que en posteriores asignaciones del bloque, algunas las nuevas interconexiones se sitúen justo en estos huecos, de forma que no destruyen información de la antigua asignación del bloque, aumentando así la ganancia en caso de que la ejecución del programa vuelva a la antigua asignación del bloque (anidaciones).

La vertiente negativa que tiene la partición máxima de la M.I. consiste en que a medida que la partición es más fina (más bloques, más registros de página, más comparadores, etc.), además del importante aumento del coste, adquiere protagonismo el algoritmo de reemplazamiento en detrimento de los registros de página, con todos los efectos que éste pueda tener sobre el comportamiento. Las ganancias obtenidas por los residuos del borrado selectivo van disminuyendo hasta llegar a su desaparición en la partición máxima. En definitiva no es muy deseable particionar en exceso la M.I. de un sistema, es preciso realizar un análisis de cual deberá ser la partición ideal para cada sistema.

Si en el mismo ejemplo que hemos desarrollado en último lugar (ejemplo III.4), aumentamos la partición de la

M.I. a ocho bloques de una posición, que sería el caso de máxima partición para un tamaño de M.I. de ocho posiciones, la evolución del contenido de los bloques será :

EJEMPLO III.5

BLOQUE	A	B	C	D	E	F	G	H
pag	1	2	3	4	5	6	7	8
p1	1	2	3	4	5	6	7	8

BLOQUE	A	B	C	D	E	F	G	H
pag	9	10	3	4	5	6	7	8
p1	9	10	3	4	5	6	7	8

BLOQUE	A	B	C	D	E	F	G	H
pag	9	10	2	3	4	5	6	7
p1	9	10	2	3	4	5	6	7

BLOQUE	A	B	C	D	E	F	G	H
pag	8	9	10	4	5	6	7	8
p1	8	9	10	4	5	6	7	8

Observamos que con el mismo programa y el mismo tamaño total de M.I. se consigue ahora una ganancia cero. El simulador da :

```

DETERPRIM          L. R. U.
NORMAL = 92  M.I. = 0  GANAN = 0.00000 %
HISTOR = 1  BLOQUES = 8 DE 1
PROGRAMA DE 11 CON LOS LAZOS :
      2 <---- 10 ( 9)

```

Si lo hacemos para el caso del Mapping Directo lo convertimos en una organizaci3n de un solo bloque de M.I. :

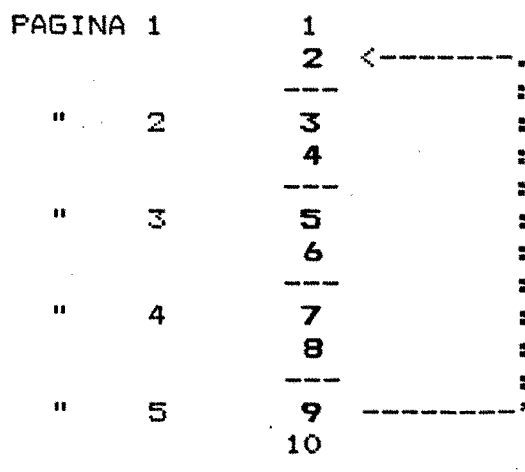
```
DETERPRIM          MAPPING DIRECTO
NORMAL = 92  M.I. = 63  GANAN = 6.84793E1 %
HISTOR = 1  BLOQUES = 8 DE 1
PROGRAMA DE 11 CON LOS LAZOS :
      2 <---- 10 ( 9)
```

Esto demuestra que con una partici3n mayor, a pesar de que se obtiene una mejor adaptabilidad del sistema al perfil del programa, no siempre es deseable llegar a un alto grado de partici3n donde con un algoritmo de este tipo (LRU) se obtienen resultados nefastos. Esto es un hecho importante, puesto que como ya se ha dicho el aumento de la partici3n supone siempre un gran aumento de coste. El acotar el grado de partici3n implica que los mayores rendimientos no ser3n siempre con los mayores costes.

Hemos de tener en cuenta que los algoritmos de reemplazamiento tienen un filtro por delante constituido por los registros de p3gina (historia), en ellos est3 contenido el n3mero de la p3gina de la memoria principal a la que pertenecia la 3ltima instrucci3n que gener3 una interconexi3n en su bloque. Cuando hay que generar una nueva interconexi3n, antes de recurrir al algoritmo, se comprueba si existe alg3n bloque cuyo R.P. haga referencia a la p3gina en

curso. Si es así, se almacena la nueva interconexión en ella. Esto tiene dos funciones, la primera de ellas corresponde al hecho de compactar de forma coherente la información en los bloques y que no se almacene de forma caótica. La segunda, la de mejorar el comportamiento de los algoritmos. Veamos que ocurriría con un algoritmo L.R.U. sin la presencia de registros de página, considerando de nuevo el caso de cuatro bloques de dos posiciones cada uno :

EJEMPLO III.6



Dado que el algoritmo es actualizado cada vez que se realiza un acceso a un bloque y que no hay registros de página, una vez se ha escrito una interconexión en un bloque, cuando se necesite escribir otra, se elegirá otro bloque para ello y la evolución de los contenidos de la M.I. vendrá dado :

pa	BLOQUE A	BLOQUE B	BLOQUE C	BLOQUE D
p1	1	x	3	x
p2	x	2	x	4

	BLOQUE A	BLOQUE B	BLOQUE C	BLOQUE D
p1	5	x	7	x
p2	x	6	x	8

	BLOQUE A	BLOQUE B	BLOQUE C	BLOQUE D
p1	9	x	3	x
p2	x	2	x	4

DETERPRIM L. R. U.

NORMAL = 42 M.I. = 0 GANAN = 0.00000 %

HISTOR = 0 BLOQUES = 4 DE 2

PROGRAMA DE 11 CON LOS LAZOS :

2 <---- 10 (4)

con lo que de nuevo obtenemos una ganancia nula, a pesar de que el lazo por su tamaño cabe en la M.I..

4.4 Algoritmo de Distancia Maxima

Dado que en principio es necesario el uso de los registros de pagina, se penso en utilizar un algoritmo de reemplazamiento que se basara en ellos, de manera que hubiera que aadir la mınima circuiteria extra para incorporar dicho algoritmo. La solucion que mejor se adapta a estos requerimientos es el algoritmo de Distancia Maxima. Este algoritmo ya ha sido descrito con anterioridad y supone solamente aadir los comparadores que determinan cual es el

bloque que contiene la página más distante de la actual. Además del ahorro de circuitería que ello supone, aporta una mejora importante del comportamiento del sistema con respecto a los anteriormente descritos.

Veamos que no existe ningún contraejemplo para este algoritmo en el que la ganancia obtenida sea cero, cuando la longitud del lazo es igual al tamaño total de M.I. más una página. Supongamos un programa de N+1 páginas con una M.I. de N bloques, y en el que la secuencia que se sigue en las páginas por las que transcurre el perfil de ejecución del mismo viene dada por :

T, T+1, T+2, T+3, T+N, T,

En este caso, la condición para que se produzca ganancia cero implicaría que al producirse un fallo de página, se eligiera, para ser reasignado, aquel bloque que fuera necesario justo a continuación del que se halla en ejecución. Si imponemos dicha condición, deberá cumplirse, al comienzo de la ejecución, que la distancia entre T y T+1 sea mayor que entre T y cualquier otro :

$$\begin{aligned}
 D(T, T+1) &> D(T, T+2) \\
 D(T, T+1) &> D(T, T+3) \\
 D(T, T+1) &> D(T, T+4) \\
 &\vdots \\
 &\vdots \\
 &\vdots \\
 D(T, T+1) &> D(T, T+N)
 \end{aligned}$$

De la misma manera :

$$\begin{array}{ll}
D(T+1, T+2) > D(T+1, T) & D(T+2, T+3) > D(T+2, T) \\
> D(T+1, T+3) & > D(T+2, T+1) \\
\vdots & \vdots \\
\vdots & \vdots \\
> D(T+1, T+N) & > D(T+2, T+N)
\end{array}$$

. . .

$$\begin{array}{l}
D(T+(N-1), T+N) > D(T+(N-1), T) \\
> D(T+(N-1), T+1) \\
\vdots \\
\vdots \\
> D(T+(N-1), T+(N-2))
\end{array}$$

$$\begin{array}{l}
D(T+N, T) > D(T+N, T+1) \\
> D(T+N, T+2) \\
\vdots \\
\vdots \\
> D(T+N, T+(N-1))
\end{array}$$

Por tanto de ahí se desprende que :

$$\begin{array}{l}
D(T, T+N) < D(T, T+1) < D(T+1, T+2) < D(T+2, T+3) < \dots \\
\dots < D(T+(N-1), T+(N-2)) < D(T+N, T+(N-1)) < D(T+N, T)
\end{array}$$

luego

$$D(T, T+N) < D(T+N, T) \tag{44}$$

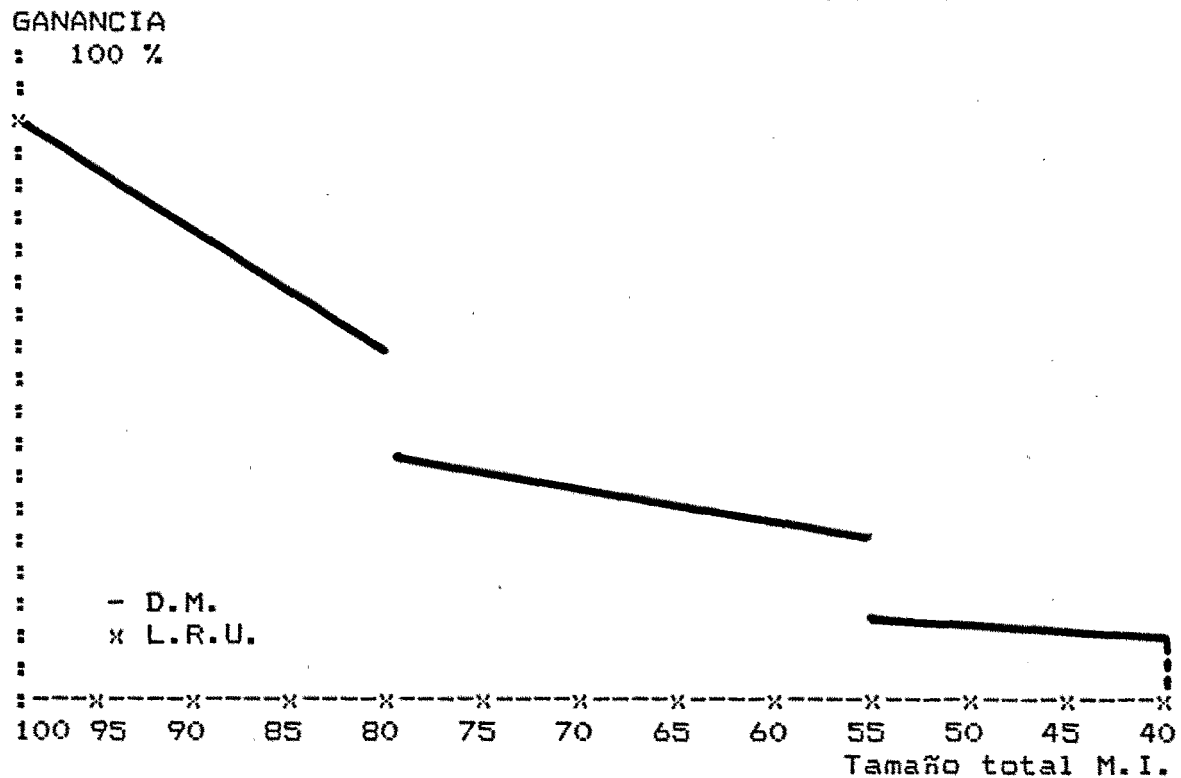
Lo cual supone una contradicción, que demuestra que este algoritmo nunca puede dar ganancia cero en el caso en que el de L.R.U. tiene ganancia cero.

Si para una partición de M.I. dada representamos las ganancias obtenidas para ambos algoritmos (L.R.U. y D.M.) en función del tamaño total de M.I., para un bucle de 100 instrucciones con 10 iteraciones y una partición en cuatro bloques, podemos ver en la gráfica III.1 como el algoritmo L.R.U. cae bruscamente a cero cuando el lazo no cabe en M.I., mientras que en el de D.M. la ganancia decrece gradualmente hasta llegar al cero.

GRAFICA III.1

LAZO = 1 <--- 100 (10) M.I. = 4 BLOQUES DE X/4 posiciones

X = Tamaño total M.I.



Los resultados que produce el algoritmo de Distancia Maxima en los casos descritos en los ejemplos anteriores y que arrojan resultados negativos para los otros algoritmos podemos resumirlos a continuacion :

En el ejemplo III.1 donde el Mapping Directo da ganancia cero porque las instrucciones ocupan las mismas posiciones en las diferentes paginas (siempre se escribe en las mismas posiciones del mismo bloque). El de Distancia Maxima obtiene el mismo resultado que el L.R.U. (ejemplo III.2) :

```
DETERPRIM          DISTANCIA MAXIMA
NORMAL = 25  M.I. = 20  GANAN = 8.00000E1 %
HISTOR = 1  BLOQUES = 2 DE 3
PROGRAMA DE 9 CON LOS LAZOS :
    7 <---- 2  (32767)
    1 <---- 8  (    5)
```

En el ejemplo III.3 donde el L.R.U. da ganancia cero, debido a que una pagina destruye a la siguiente en ejecucion, el algoritmo de Distancia Maxima tiene una ganancia igual que la de Mapping Directo :

```
DETERPRIM          DISTANCIA MAXIMA
NORMAL = 101  M.I. = 54  GANAN = 5.34653E1 %
HISTOR = 1  BLOQUES = 4 DE 2
PROGRAMA DE 11 CON LOS LAZOS :
```


1 <---- 10 (9)

En el ejemplo III.4 igual que el anterior pero con un lazo menor, la ganancia obtenida por L.R.U. no es cero pero mucho menor que la obtenida por Mapping Directo y Distancia Máxima :

```
DETERPRIM          DISTANCIA MAXIMA
NORMAL = 92  M.I. = 63  GANAN = 6.84783E1 %
HISTOR = 1  BLOQUES = 4 DE 2
PROGRAMA DE 11 CON LOS LAZOS :
```

2 <---- 10 (9)

En el caso de partición máxima de la M.I., en el ejemplo III.5, el L.R.U. da ganancia cero mientras que el de Distancia Máxima al igual que el Mapping Directo :

```
DETERPRIM          DISTANCIA MAXIMA
NORMAL = 92  M.I. = 63  GANAN = 6.84783E1 %
HISTOR = 1  BLOQUES = 8 DE 1
PROGRAMA DE 11 CON LOS LAZOS :
```

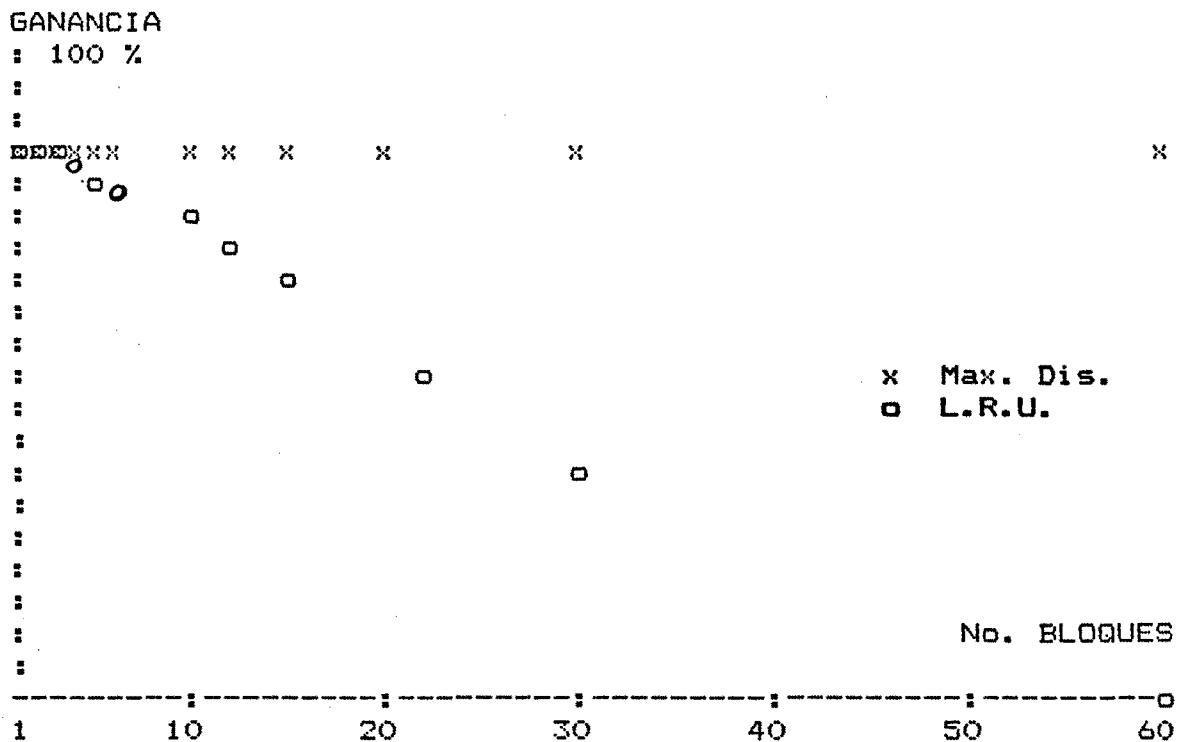
2 <---- 10 (9)

4.5 Comparación de los algoritmos L.R.U. y D.M.

Para visualizar el comportamiento del algoritmo de Distancia Máxima con respecto al L.R.U., se ha desarrollado un ejemplo. En dicho ejemplo el programa simulado es un lazo

de 61 instrucciones que tiene 9 vueltas atrás en su bifurcación terminal, es decir se ejecuta 10 veces. Se ha tomado un tamaño total de M.I. de 60 posiciones y se ha ido particionando en sucesivas simulaciones. Se presenta una curva de cual es la ganancia en función del número de bloques que tiene cada partición :

GRAFICA III.2



Con estos resultados podemos ver que el algoritmo de Distancia Máxima es bastante insensible a la partición, a pesar de que el tamaño de la M.I. es un poco inferior al del lazo. Por el contrario, en el caso del L.R.U. a medida que aumenta la partición los registros de página pierden protagonismo y lo adquiere el algoritmo, hasta llegar al caso extremo de partición máxima donde estamos en una situación crítica en la que los R.F. no actúan y el tamaño de M.I. es

inferior al del lazo justo en un bloque, situación en la cual dicho algoritmo produce ganancia cero como ya se ha visto.

En consecuencia vemos que en los casos estudiados, el algoritmo de Distancia Máxima supera a los otros dos en los diferentes casos planteados. Creemos pues que la elección de este algoritmo para su incorporación al sistema es acertada.

III.5 EJEMPLO PARA LA COMPARACION DE LOS METODOS CLASICOS CON EL PROPUESTO

Con objeto de valorar la ganancia que se obtiene en un sistema con la incorporación del coprocesador, con respecto al caso off-line, se ha desarrollado un ejemplo sobre el que se han realizado mediciones para ambos métodos.

El sistema que se ha elegido para este ejemplo es un ordenador microprogramable HP21MX y el programa que se ha analizado :

START	CLA		SAX DATA		CMA, INA
	STA SAVE		LDA POS		ADA B1
	CAX		SAX DATA+1		SZA, RSS
SAND	LAX DATA		LDA B1		JMP START
	STA POS		STA SAVE		CLA
	CMA, INA	NCAMB	ISX		CAX
	STA MD		CXA	ALMAC	LAY CONT+1
	LBX DATA+1		CMA, INA		CYA
	ADB MD		ADA CONT		STA VARI
	SXB, RSS		SZA	INDE	LDA VARI
	JMP NCAMP		JMP SAND		ARS
	LAX DATA+1		LDA SAVE		STA VARI

LAX PAPE		ADA VARI		SAX DIMO
CMA, INA		CAY		JMP X2
STA M1		JMP X1	X3	LDA EXI
LBY DATA	REF	CLE		SAX DIMO
ADB M1		CYA	X2	ISX
SZB, RSS		ADA VARI		CXA
JMP X3		CAY		CFA CONT+2
SEZ, RSS	X1	LDA VARI		JMP X4
JMP REF		SZA		JMP ALMAC
CYA		JMP INDE	X4	HLT
CMA, INA		LDA FALL		END

El grafo correspondiente al programa descrito aparece en la figura III.4. El número de instrucciones que tiene cada segmento "aj", la frecuencia con que se ejecuta y la probabilidad de salto que por tanto le corresponde a la bifurcación terminal de los segmentos :

Segmento	No. instr.	F.i	P.i
a1	3	16.2	
a2	7	110.2	0.2976407
a3	1	77.4	1
a4	6	32.8	
a5	5	110.2	0.1470055
a6	1	94	1
a7	4	16.2	0.0617284
a8	1	15.2	1
a9	2	1	
a10	3	3.4	
a11	9	11.4	0.8596491
a12	1	1.6	1
a13	1	9.8	0.6734693
a14	1	3.2	1
a15	5	6.6	1
a16	4	3.2	

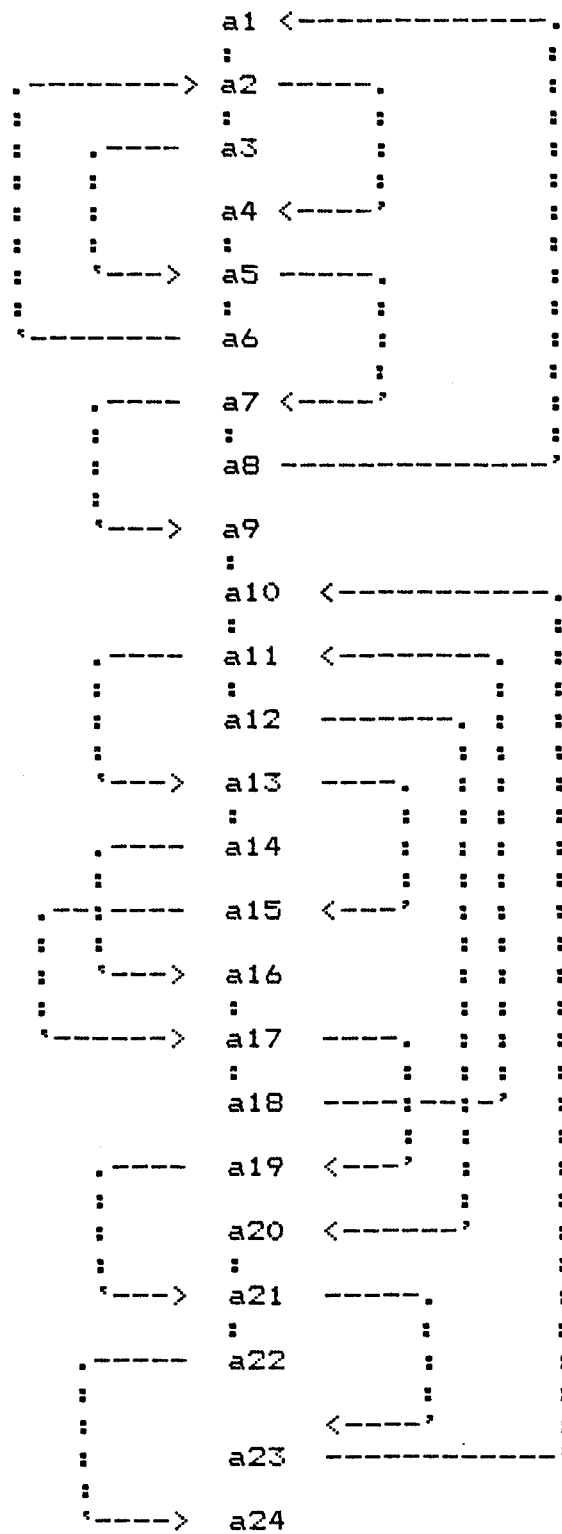


Figura III.4 Grafo correspondiente al programa ejemplo

Segmento	No. instr.	F.i	P.i
a17	2	9.8	0.1836734
a18	1	8	1
a19	3	1.8	1
a20	2	1.6	
a21	3	3.4	0.7058823
a22	1	1	1
a23	1	2.4	1
a24	1	1	

Este ejemplo fue desarrollado en el caso off-line por (Ri80) y teniendo en cuenta que se realizó la migración de la totalidad del programa, sin utilizar factor de compactación (F=1). Los resultados que se obtuvieron fueron :

Tiempo total de ejecución = 6080.46 microseg.

Tiempo total de ejecución en versión migrada = 2801.49 microseg.

Lo que supone una reducción del tiempo de ejecución al 46% del original.

En el caso on-line, utilizando una M.I. de dos bloques cuyo tamaño es igual al número de palabras necesarias para albergar 20 instrucciones cada uno, es decir con capacidad para capturar el lazo más grande, y por tanto todos los lazos, se obtiene que el 96,75 % de las instrucciones se ejecutan en su versión migrada. Esto supondría que

el tiempo de ejecución sería de 2913.29 microsegundos, de modo que el tiempo de ejecución se reduce al 47.9%. Este resultado se ha obtenido con el simulador probabilístico "BRANCLRU" ejecutando el programa simulado 500 veces. El resultado obtenido por el simulador :

BRANCLRU DISTANCIA MAXIMA
 NORMAL = 2.045219E3 M.I. = 1.978643E3 GANAN = 9.67529E1
 HISTOR = 1 BLOQUES = 2 DE 20
 PROGRAMA DE 68 CON LOS LAZOS :

12	<---	10	(2.97641E-1)	(0)
18	<---	11	(1.00000)	(32767)
24	<---	22	(1.47006E-1)	(0)
4	<---	23	(1.00000)	(32767)
29	<---	27	(6.17284E-2)	(0)
1	<---	28	(1.00000)	(32767)
44	<---	42	(8.59649E-1)	(6)
61	<---	43	(1.00000)	(32767)
46	<---	44	(6.73469E-1)	(2)
51	<---	45	(1.00000)	(32767)
55	<---	50	(1.00000)	(32767)
58	<---	56	(1.83673E-1)	(0)
34	<---	57	(1.00000)	(32767)
63	<---	60	(1.00000)	(32767)
67	<---	65	(7.05882E-1)	(2)
68	<---	66	(1.00000)	(32767)
31	<---	67	(1.00000)	(32767)

III.6 OBTENCION DE UN MODELO ANALITICO DE COMPORTAMIENTO

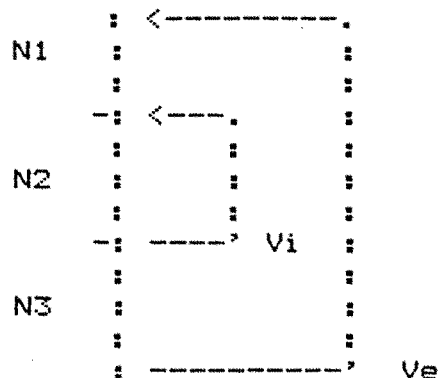
Los programas de simulación que han sido presentados tienen como un primer objetivo analizar el comportamiento del sistema bajo ciertos conjuntos de parámetros del mismo, para determinar cual es la ganancia obtenida por el sistema al ejecutar determinados tipos de programas, incorporando el coprocesador para migración vertical en tiempo de ejecución. El segundo objetivo que se plantea, para ser desarrollado mediante el simulador, consiste en hallar un modelo algorítmico de comportamiento del sistema que permita de forma teórica determinar la ganancia conseguida al ejecutar determinado tipo de programas y así facilitar la elección de la configuración más apropiada, en cuanto a tamaño y partición de M.I., según el tipo de programas que se pretenda ejecutar en un sistema determinado. En esta línea se ha tratado de modelizar la estructura de control de ejecución de los programas puesto que no existen, desarrollados, modelos en tal sentido.

6.1 Reducción al lazo simple

Puesto que la forma de actuación del sistema presentado, es básicamente local, es decir, se obtienen ganancias en velocidad siempre que se pueda capturar un bucle en M.I., con independencia del tipo de programa al que pertenezca dicho lazo, nuestro estudio se ha particularizado en este tipo de estructuras, de modo que muchas de las

estructuras repetitivas se reducen a un lazo. Por ejemplo una estructura formada por bucles anidados, puede ser reducida a dos lazos simples e independientes.

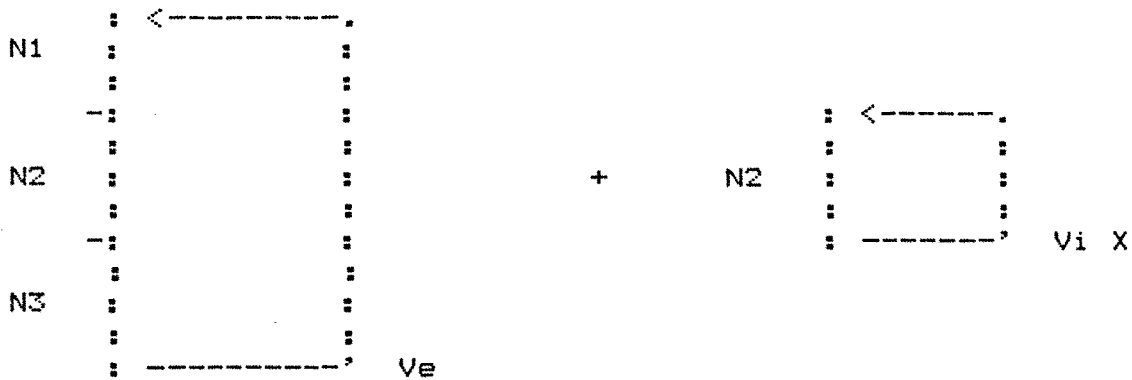
Sea un programa formado por los siguientes bucles anidados :



Donde "N2" es el número de instrucciones del lazo interno, "N1" y "N3" el número de instrucciones que pertenecen al externo pero no al interno. "Vi" es el número de vueltas atrás que debe realizar la bifurcación terminal del lazo interno y "Ve" el que debe realizar la del externo. El número total de instrucciones ejecutadas en este caso será :

$$A = (Ve + 1) (N1 + N3 + N2 (Vi + 1)) \quad (45)$$

Se pretende reducir la estructura con anidación al caso de dos lazos independientes encadenados consecutivamente :



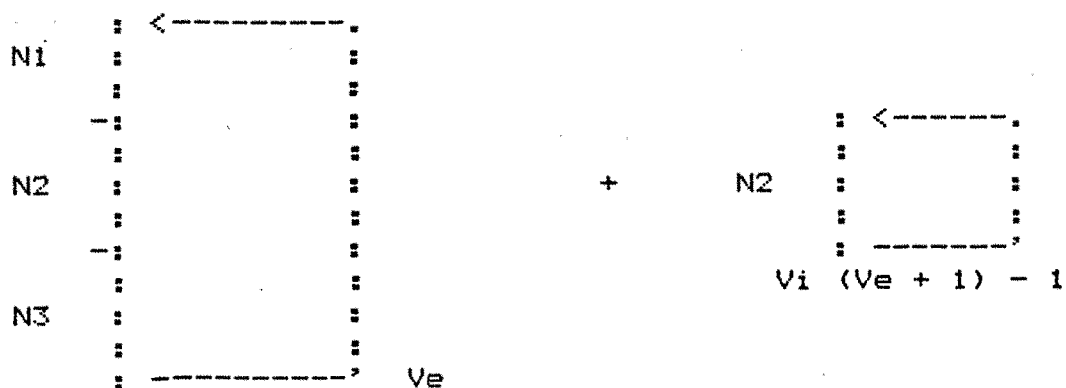
Nuestra pregunta es, ¿cuanto debe valer "X" para que el número de instrucciones ejecutadas sea el mismo que en la situación inicial?

$$B = (N1 + N2 + N3) (Ve + 1) + N2 (Vi X + 1) \quad (46)$$

Igualando las expresiones correspondientes a "A" y "B" se obtiene que el valor de "X" debe ser :

$$X = \frac{(Ve + 1) Vi - 1}{Vi} \quad (47)$$

Quedando de esta forma :



De forma recursiva puede extenderse esta idea a cualquier estructura que contenga diferentes niveles de anidación de lazos, reduciendo el problema del estudio del comportamiento del sistema sobre este tipo de estructuras, al estudio de lazos simples, independientes. Se ha comprobado con el simulador, para diversos ejemplos, y tomando los casos extremos de partición de M.I., es decir, un solo bloque o bloques de una sola posición, que el número de instrucciones ejecutadas en versión "J+1/2" para el caso de la estructura inicial anidada y para el caso de los lazos equivalentes, coincide. Esta comprobación se ha hecho usando el algoritmo de Distancia Máxima y partiendo de un tamaño de M.I. igual al del lazo exterior, reduciendo dicho tamaño en sucesivas simulaciones. También se ha comprobado que la ganancia obtenida, es independiente de la posición relativa del lazo interior respecto del exterior, dado que las partes que se puedan destruir, en M.I., por el hecho de desplazar el lazo interior, sólo deja de encontrarse en una primera vuelta del lazo interior. Dentro de esta idea de reducir el estudio de las estructuras repetitivas al lazo simple, faltarían por estudiar aquellas estructuras de longitud variable, como las generadas por el tipo "IF ... THEN ...", en las que en función de una determinada condición, varía el número de instrucciones que se ejecutan en una iteración. Su estudio requiere un análisis estadístico especial, por lo que ha sido incluido como un objetivo en futuras ampliaciones de este trabajo.

6.2 Comportamiento del sistema. El modelo analítico

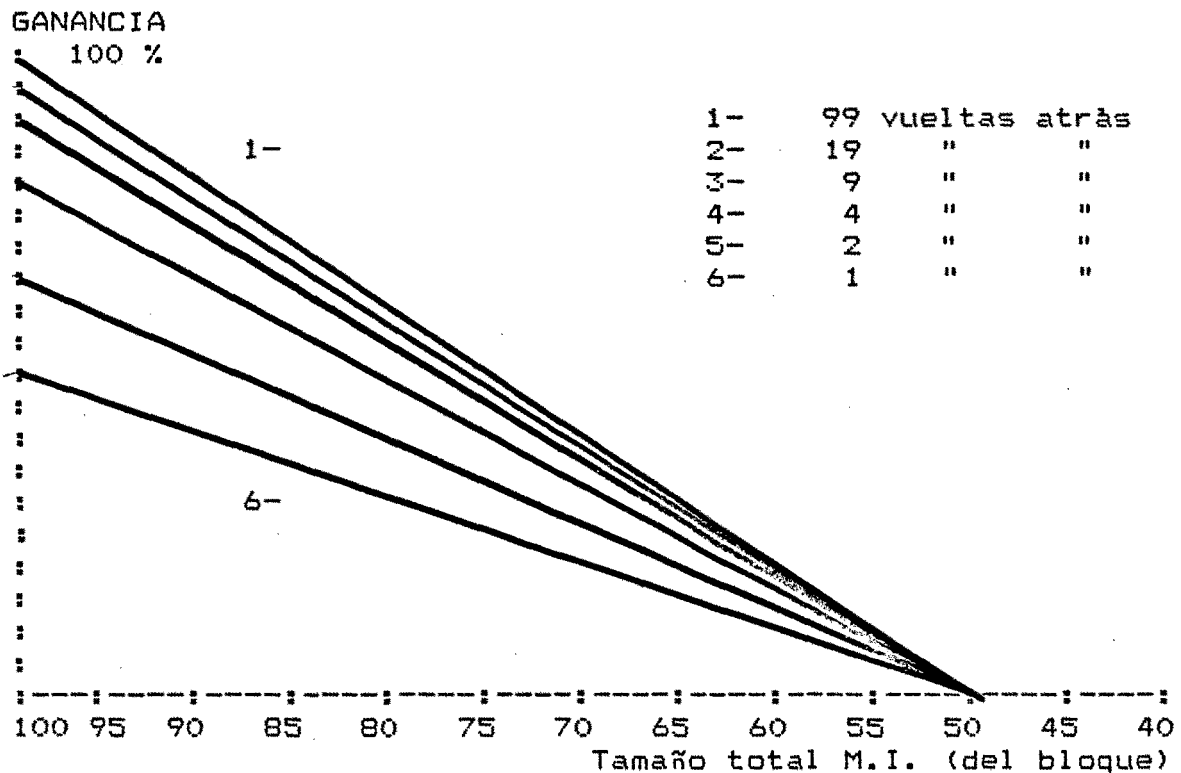
Con objeto de realizar el estudio del comportamiento del sistema, se ha analizado dicho comportamiento para el algoritmo de Distancia Máxima, tomando un lazo estandar de cien instrucciones, calculando las ganancias que se obtienen para diferentes tamaños totales de la M.I., diferentes particiones y diferentes números de iteraciones en el lazo. El resultado de dicho estudio ha producido una serie de curvas de la ganancia obtenidas en función del tamaño y partición de M.I..

Para el caso en que la M.I. esté constituida por un único bloque, se muestra la curva experimental obtenida con el simulador en la gráfica III.3.

Las diferentes curvas que se representan, corresponden a diferentes números de iteraciones en el lazo, la superior pertenece a un lazo de cien instrucciones que se ejecuta con 100 iteraciones, es decir 99 vueltas atrás, las siguientes hacia abajo corresponden a los números de iteraciones : 20, 10, 5, 3 y 2 respectivamente, lo cual supone, en vueltas atrás : 19, 9, 4, 2 y 1. En cuanto al tamaño del bloque se parte de un valor inicial igual al tamaño del lazo (100), puesto que para tamaños superiores de M.I. la ganancia no sufre modificación respecto de este caso.

GRAFICA III.3

LAZO = 1 <--- 100 (z) M.I. = 1 BLOQUE DE x posiciones



En esta gráfica podemos ver que todas las curvas forman un haz de rectas de diferente pendiente, en función del número de iteraciones, y que todas ellas pasan por el punto de coordenadas (50 , 0). Este punto corresponde al tamaño del bloque de M.I. por debajo del cual la ganancia obtenida es cero, con independendencia del número de iteraciones.

Para analizar este resultado supongamos el caso crítico en que el tamaño del bloque de M.I. es de cincuenta posiciones.

Al iniciarse la ejecución del lazo, el bloque será

completado con las interconexiones de la primera mitad del lazo y al ejecutar la segunda mitad del mismo, las interconexiones de ésta serán escritas sobre las de la otra mitad, por ello en las sucesivas iteraciones no se encontrará nunca ninguna interconexión en M.I., ganacia cero. Supongamos ahora que el bloque tiene una posición más, es decir 51.

Al ejecutar por primera vez el lazo, y rellenar el bloque, en él se insertarán 51 interconexiones, ello significa que la interconexión correspondiente a la instrucción 52 se almacenará en la posición 1 del bloque y que al finalizar la primera iteración del lazo, el contenido del bloque será :

Dirección posición en el bloque	Dirección instrucción de la interco- nexión
1	52
2	53
.	.
.	.
.	.
48	99
49	100
50	50
51	51

Cuando la ejecución inicie la segunda iteración, la interconexión correspondiente a la instrucción "1" volverá a ser escrita en la posición "1" del bloque y así sucesivamente hasta llegar a la "50", donde en ella y en la siguiente se encontrarán las interconexiones en M.I.. Esto ocurrirá en todas las iteraciones del lazo por lo que el

número de instrucciones ejecutadas en versión "J+1/2" será :

$$NUINMI = 2 V$$

Donde "V" es el número de vueltas atrás en la bifurcación terminal. La ganancia vendrá dada por :

$$G = \frac{2 V}{100 (V + 1)} \quad (48)$$

Cada vez que aumentamos el tamaño del bloque en uno, el número de instrucciones ejecutadas en versión "J+1/2" aumenta en dos, en general si "TL" es el tamaño del lazo (100) y "TB" el del bloque, el número de instrucciones en "J+1/2" será :

$$NUINMI = V (2 TB - TL) \quad (49)$$

Para calcular la ganancia debemos dividir "NUINMI" por el número total de instrucciones ejecutadas "TL (V + 1)", quedando la ganancia para una M.I. de un solo bloque :

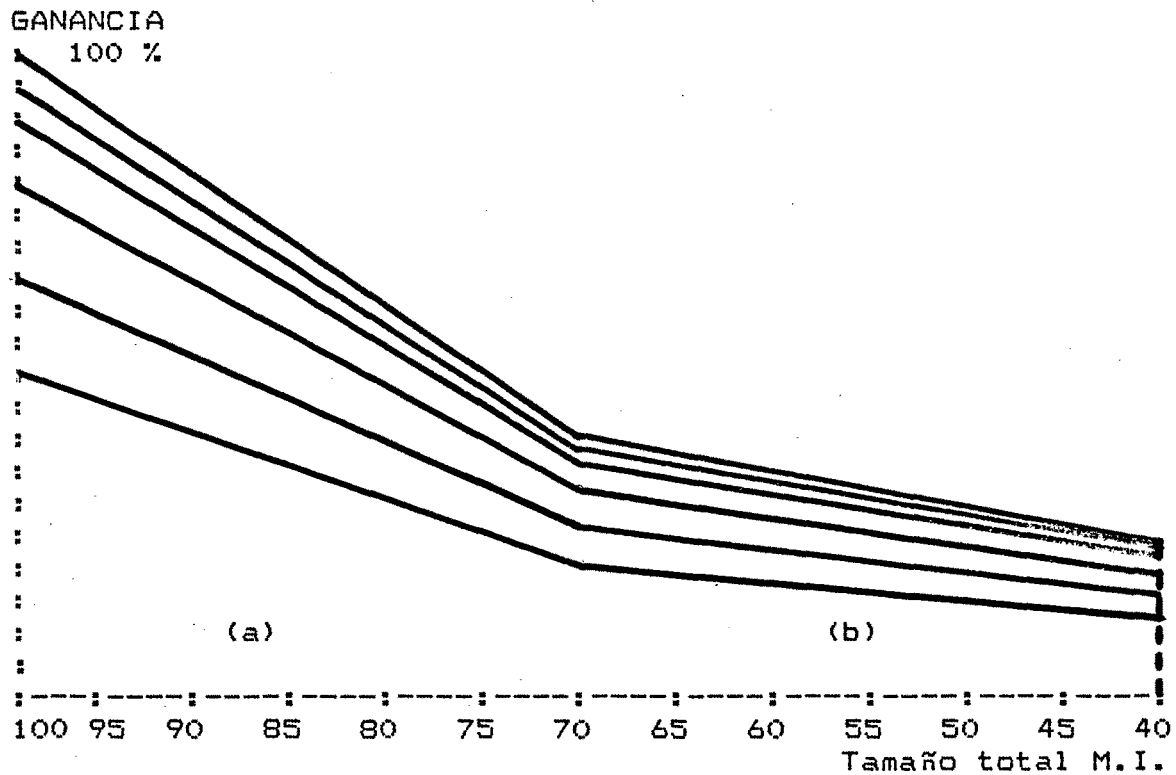
$$G = \frac{V}{V + 1} \left(2 \frac{TB}{TL} - 1 \right) 100 \% \quad (50)$$

Siendo esta la expresión teórica del haz de rectas, hallado experimentalmente por el simulador (Grafica III.3).

Veamos ahora el resultado experimental del mismo estudio, en el caso de que la M.I. esté particionada en dos bloques :

GRAFICA III.4

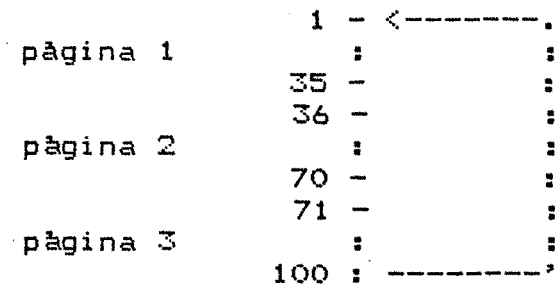
LAZO = 1 <--- 100 (z) M.I. = 2 BLOQUES DE x posiciones



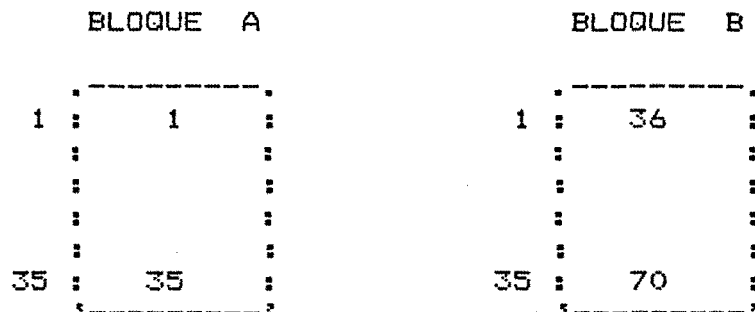
En este caso, cada una de las curvas correspondientes a un número de iteraciones, está formada por dos rectas de diferente pendiente. El tramo de la izquierda (a), se puede ver que es idéntico al obtenido en la gráfica III.3, con la pendiente ya determinada, y la parte derecha (b) con una pendiente menor.

Para estudiar este caso supongamos inicialmente la situación en que el tamaño total de M.I. es de 70. En este

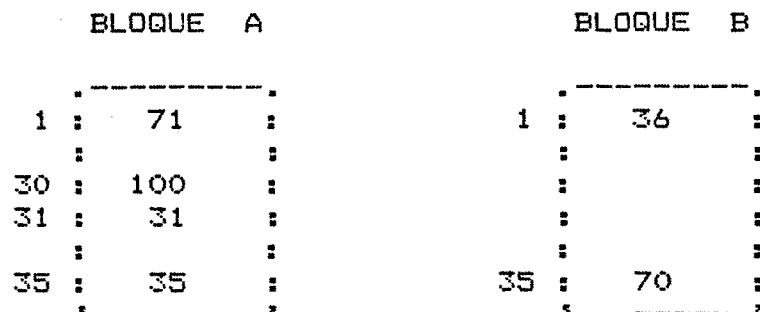
caso la memoria principal se verá virtualmente en páginas de 35 posiciones :



La forma en que se irán rellorando los dos bloques durante la primera iteración será :

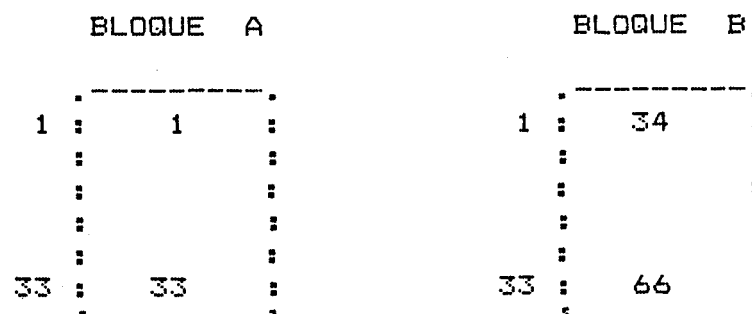
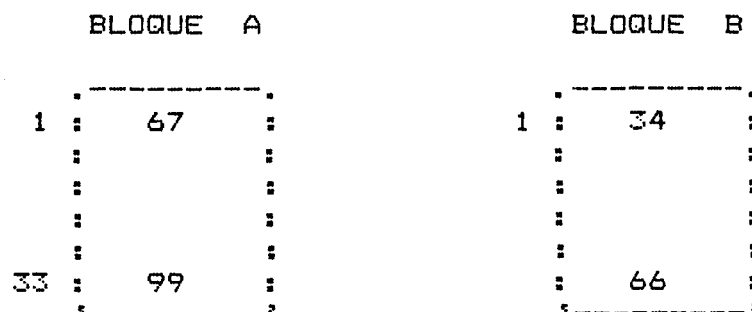
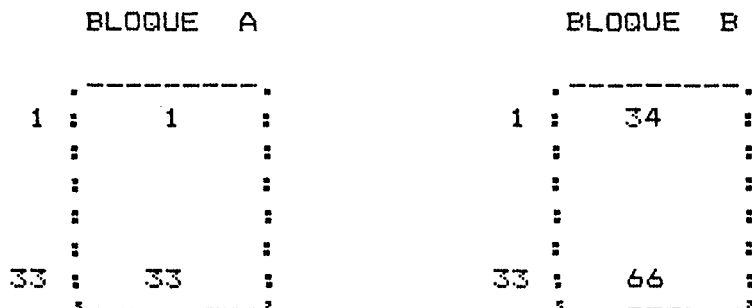


Para almacenar las interconexiones de la siguiente página, el algoritmo de Distancia Máxima seleccionará el bloque "A" que es aquel cuyo contenido es más lejano de la tercera página.



Al iniciarse la segunda iteración, el algoritmo D.M. deberá seleccionar un bloque para almacenar la página "1", que no está en M.I.. El bloque elegido será el "A", puesto que la página que ahora contiene es la "3", que es la más lejana de la "1". En esta situación cuando llegue a la instrucción "31" encontrará su interconexión, y así hasta llegar a la número "70". Este proceso se repetirá indefinidamente hasta completar todas las iteraciones del lazo, de manera que salvo las instrucciones de la "1" a la "30" y de la "71" a la "100", los contenidos de los bloques permanecen estacionarios, generando por ello ganancia en velocidad. Fijémonos que de la "31" a la "35" las encontramos gracias a los residuos producidos por el borrado selectivo de M.I., pero de la "36" a la "70" se encuentran por el hecho de que el bloque "B" no ha sido seleccionado en ningún momento por el algoritmo de D.M.. Advertamos que en esta situación el proceso bajo el que se van llenando los bloques es igual que si se tratara de un único bloque, esto justifica que en esta zona (a) de la gráfica III.4 la pendiente de las rectas coincida con el caso de M.I. sin partición. El punto donde se produce la ruptura en la gráfica (de una pendiente a la otra), corresponde a la situación en que el tamaño de la M.I. es tal que los residuos por el borrado selectivo ya no pueden permanecer en el bloque "A" y de este modo la ganancia sólo se produce por el hecho de que el bloque "B" no sea seleccionado. El punto crítico de ruptura por tanto corresponderá al caso en que el tamaño del bloque "A" sea igual a

un tercio del tamaño del lazo, es decir, cuando el tamaño de M.I. sea los dos tercios del tamaño del lazo. Para verlo con exactitud supongamos ahora que el lazo tiene una longitud de 99 instrucciones :



A partir de este valor del tamaño de M.I., el número de instrucciones ejecutadas en versión "J+1/2" deberá ser calculado como :

$$NUINMI = V TB \quad (51)$$

La ganancia obtenida para la zona (b) de la gráfica III.4 vendrá expresada como :

$$G = \frac{V \text{ TB}}{(V + 1) \text{ TL}} 100 \% \quad (52)$$

Para visualizar mejor el comportamiento del sistema, consideremos el programa simulado no ya por el número de instrucciones que contiene, sino por el número de páginas en que queda particionado para una configuración de M.I. dada, pero siempre partiendo del lazo de 100 instrucciones. Volvamos atrás y analicemos el caso de en que la M.I. tenía un tamaño total de 70 y estaba particionada en dos bloques, es decir nos encontramos de nuevo en la zona de la izquierda de la gráfica III.4. En tal situación, el lazo estará particionado en tres páginas, la tercera de las cuales no está completa, sino que sólo abarca una porción de la página dejando libre otra porción "D" de la misma, donde:

$$D = \text{TB} - (\text{TL} \text{ mod } \text{TB}) \quad (53)$$

Si observamos el comportamiento del algoritmo, veremos la evolución del contenido de los dos bloques y podremos determinar la ganancia :

BLOQUE A 1 3 1 3 1 3 1 . . . páginas

BLOQUE B 2 2 2 2 2 2 2 . . . páginas

Vemos que el bloque "B" mantiene su contenido y que por ello produce un

$$\text{NUINMI}_B = V \text{ TB} \quad (54)$$

El bloque "A" va cambiando de contenido pero la porción "D" de éste permanece invariante, proporcionando por ello un

$$\text{NUINMI}_A = V \text{ D} \quad (55)$$

El "NUINMI" total obtenido será la suma de los dos parciales que acabamos de calcular :

$$\text{NUINMI} = V (\text{TB} + \text{D}) \quad (56)$$

Si no situamos ahora en la zona derecha de la gráfica III.4, con una M.I. de 60 posiciones particionada en dos bloques, el programa ocupará cuatro páginas, la última de las cuales tendrá un hueco vacío $D = 20$. La evolución del contenido de estos bloques será :

BLOQUE A	1	3	3	3	3	3
BLOQUE B	2	4	1	2	4	1

Una vez la ejecución ha terminado la primera iteración del bucle, los bloques contienen las páginas "3" y

"4" respectivamente, al introducir la página "1" de nuevo, el bloque cuyo contenido es más distante es el que contiene la página "4" (el "B"), de modo que la "3" permanece invariante. A la hora de introducir la página "2" existe equidistancia de ésta con respecto de los dos bloques, puesto que éstos contienen las páginas "1" y "3". El algoritmo de D.M. en estos casos resuelve seleccionando aquella que tiene un menor valor, dado que normalmente la ejecución de un programa progresa hacia valores mayores del contador de programa, y la esperanza de respetar información necesaria posteriormente es mayor. Ello hace que la página "2" sustituya a la "1" y así la "3" continúe en el bloque "A", generando la consiguiente ganancia de un bloque. Es claro que en esta situación no se produce ninguna ganancia por los residuos puesto que estos pertenecen a un bloque que es sustituido sistemáticamente. En este caso por tanto

$$NUINMI = V TB \quad (57)$$

El valor de NUINMI se hará cero cuando no exista la posibilidad de que un bloque permanezca invariante. Esta posibilidad se mantiene mientras el programa esté paginado en cuatro páginas o menos, pero si lo es en más, entonces desaparece la equidistancia y la ganancia se va a cero, lo que concuerda perfectamente con la gráfica experimental hallada.

BLOQUE A	1	3	5	:	1	3	5
				:			
				:			
BLOQUE B	2	4	4	:	2	4	4

Si queremos aplicar este modo de analizar el comportamiento del sistema en el caso en que la M.I. sea de un solo bloque, dado que en este caso no existe la posibilidad de un bloque mantenga su contenido invariante en el tiempo, para calcular NUINMI, deberemos considerar sólo la incidencia de los residuos.

$$NUINMI = V D$$

$$NUINMI = V (TB - (TL \text{ mod } TB)) \quad (58)$$

que coincide con el valor determinado en (49).

Para determinar el comportamiento del sistema para configuraciones de M.I. con particiones de tres y más bloques, podemos aplicar el mismo método a las mismas.

La forma de conseguir ganancia en el sistema se reduce a las dos fuentes de ganancia que se han visto, por invariancia del contenido de un bloque o más, y por los residuos en un bloque. En general, la ganancia por los residuos sólo aparece cuando el tamaño del lazo, en páginas ocupadas, supera en una página al tamaño total (TMI) de la M.I., es decir :

$$TMI \leq TL < TMI + TB \quad (59)$$

En estas condiciones si tenemos "NB" bloques de M.I., la evolución de los contenidos de los bloques en la primera iteración será :

BLOQUE	A	1	NB+1
BLOQUE	B	2	2
	.		
	.		
	.		
BLOQUE	NB	NB	NB

Durante la segunda iteración :

BLOQUE	A	1	NB+1
BLOQUE	B	2	2
	.		
	.		
	.		
BLOQUE	NB	NB	NB

y así sucesivamente. Se observa, por tanto que en el primer bloque ("A") las páginas "1" y "NB+1" se van sustituyendo la una a la otra, pero la página "NB+1" no destruye a la "1" en su totalidad, sino que respeta la porción final ("D") de la misma, los residuos. Esto producirá un NUINMir :

$$\text{NUINMir} = V D \quad (60)$$

Si disminuimos el tamaño de M.I. por debajo del límite inferior fijado, el efecto de los residuos desaparece.

Nótese que los demás bloques mantienen invariantes sus contenidos con lo que se obtiene un NUINMIb :

$$\text{NUINMIb} = V \text{ TB} \quad (\text{NB} - 1) \quad (61)$$

El "NUINMI" total será la suma de ambos (NUINMIr y NUINMIb). En cualquiera de los casos estudiados, el cálculo de la ganancia se obtiene :

$$G = \frac{\text{NUINMI}}{(V + 1) \text{ TL}} \quad (62)$$

El límite superior de ganancia lo constituye pues, este caso en el que este se produce por la suma de los dos efectos, residuos y bloques estacionarios. El límite inferior, viene marcado en el caso de que el número de páginas que ocupa total o parcialmente el programa (NP), sea igual a :

$$\text{NP} = 3 \text{ NB} - 1 \quad (63)$$

Para demostrarlo, sea un lazo con "NP" páginas con una M.I. particionada en "N" bloques (NP es el cociente por exceso entre TL y TB). La evolución que seguirán los contenidos de los bloques durante la primera iteración, deberá ser :

BLOQUES	1	2	. . .	N-1	N
pàginas	1	2	. . .	N-1	N
"	N+1	N+2	. . .	2N-1	2N
"	2N+1	2N+2	. . .	3N-1	

Durante la segunda iteraci3n ocurrirà :

BLOQUES	1	2	. . .	N-1	N
Pag. antiguas	2N+1	2N+2	. . .	2N-1	2N
pag. nuevas	N-1	N-2		1	

La pàgina "N" deberà ser colocada en el bloque "N", puesto que es el que tiene el contenido mäs alejado. Por ello, se produce la destrucci3n de toda la informaci3n antigua de M.I. al iniciar la segunda iteraci3n :

BLOQUES	1	2	. . .	N-1	N
	N-1	N-2	. . .	1	N

Veamos ahora lo que ocurre por encima de este limite inferior ($3N - 1$), y por debajo del limite superior que hemos definido. Para ello dividimos el problema en las siguientes zonas :

- a- $NP \leq NB$
- b- $NB < NP \leq 2NB$
- c- $2NB < NP < 3NB - 1$

a- En este caso el lazo cabe completamente en el

interior de M.I. y el "NUINMI" que se produce será igual a :

$$\text{NUINMI} = V \cdot \text{TL} \quad (64)$$

b- Aquí el lazo ya no cabe por completo en M.I. y debemos ver cual es la evolución de los contenidos de los bloques durante las iteraciones del lazo. Veamos un ejemplo donde NP = 18 y NB = 10 :

En la primera vuelta :

BLOQUES	1	2	3	4	5	6	7	8	9	10
	1	2	3	4	5	6	7	8	9	10
	11	12	13	14	15	16	17	18		

En la segunda :

BLOQUES	1	2	3	4	5	6	7	8	9	10
	1	2	3	4	5	6	7	8	9	10
	11	12	13	14	15	16	17	18		
			6	5	4	3	2	1		
						8	7			

Vemos aquí que se produce ganancia al encontrarse en M.I. las páginas : 9, 10, 11 y 12. Separemos el motivo por el que se gana en 9 y 10 del de 11 y 12, puesto que 9 y 10 son encontrados porque durante la primera iteración no son destruidos, y en cambio 11 y 12 son encontrados porque no son destruidos durante la segunda iteración.

Si pretendemos generalizar este tratamiento para cualquier N y NP (en el caso b-), y consideramos un lazo con $NP = 2N - X$ ($0 \leq X < N$), el motivo por el que se encontraron 9 y 10 hará que se encuentren "Q1" bloques :

$$Q1 = X \quad (65)$$

mientras que el motivo por el que se encontraron 11 y 12 se basa en que al ir reasignando los bloques, durante la segunda iteración, de derecha a izquierda, podemos encontrarnos una página que se halle más alejada de la página "1" que de la que le correspondería al ir de derecha a izquierda. El algoritmo D.M. irá asignando bloques de derecha a izquierda hasta que una página esté a la misma o mayor distancia del "1" que de la que ordinalmente le corresponde.

BLOQUES	1	2	3	.	.	.		N-2	N-1	N
	1	2	3	.	.	.		N-2	N-1	N
	N+1	N+2	N+3	.	.	.	2N-X			
				.	.	.	5 4 3 2 1			

Si contamos los bloques al revés de su numeración inicial, es decir de derecha a izquierda mediante un puntero "Y", vemos que el "1" irá debajo de "2N-X", el "2" debajo de "2N-X-1" ... y el "Y-X" debajo de "2N-(Y-1)". El bloque "Y" con contenido "2N-(Y-1)" quedará invariante si la página "Y-

X" está más alejada de la "1" que de ésta :

$$(2N - (Y - 1)) - (Y - X) \leq (Y - X) - 1$$

$$Y \geq \frac{2X + 2N + 2}{3} \quad (66)$$

es decir, el cociente por exceso. Resumiendo "Y" señala el primer bloque (contando de derecha a izquierda) cuyo contenido permanece invariante. El número "Q2" de bloques invariantes estará formado por este bloque y todos los que se hallen a su izquierda, siempre que el valor de "Y" sea menor que "N".

$$Q2 = N - Y + 1 \quad \text{si } Y \leq N \quad (67)$$

$$Q2 = 0 \quad \text{si } Y > N$$

El número total de bloques invariantes en este caso "b" será :

$$Q = Q1 + Q2 \quad (68)$$

En este apartado, en el caso particular en que $NP=NB+1$, se producirá el efecto de los residuos y por ello durante el cálculo de "NUINMI" deberemos tenerlo en cuenta.

c- En este tercer caso desaparecen los bloques que mantenían invariante su contenido por no ser destruido

durante la primera iteración (caso 9 y 10 del apartado anterior). Elijamos un ejemplo en el que $NP = 3N-X$ ($1 < X < N$), durante la primera iteración :

BLOQUES	1	2	.	.	.		N-1	N
	1	2	.	.	.		N-1	N
	N+1	N+2	.	.	.		2N-1	2N
	2N+1	2N+2	.	.	.		3N-X	

Durante la segunda iteración :

BLOQUES	1	2	.	.	.		N-1	N	
	1	2	.	.	.		N-1	N	
	N+1	N+2	.	.	.		2N-1	2N	
	2N+1	2N+2	.	.	.		3N-X		
	N-X	N-X-1	.	.	.		3	2	1
									N-X+1

Hasta llegar a asignar la página "N-X", inclusive, nunca puede darse la posibilidad de que ninguna vaya debajo del "1", puesto que la distancia de "N-X" a "2N+1" es mayor que de "N-X" a "1". Para seleccionar un bloque para la página "N-X+1" tengamos en cuenta que su distancia a "1" es N-X y su distancia a 2N es N+X-1 y como $1 < X < N$, la página N+X-1 deberá ir debajo de 2N. El primer bloque que quedará invariante será aquel cuya página candidata sea colocada debajo de "1", en lugar de en él.

En el bloque "Y-esimo" (contando desde la derecha)

el contenido antiguo será la página $2N-(Y-1)$ y el que se almacenará de nuevo $N+Y-X$. La condición para encontrar el primer bloque que permanece invariante es :

$$(2N - (Y - 1)) - (N + Y - X) \leq (N + Y - X) - 1$$

$$Y \geq \frac{2X + 2}{3} \quad (69)$$

cociente por exceso. Todos los bloques a la izquierda de éste, hasta el anterior al que contiene el "1", mantendrán su contenido. El bloque que contiene la página "1" ocupa la posición $Y("1") = X+1$. El número de bloques invariantes "Q" será :

$$Q = Y("1") - Y = X + 1 - Y$$

$$Q = X + 1 - Y \quad \text{si } Y("1") > Y \quad (70)$$

$$Q = 0 \quad \text{si } Y("1") \leq Y$$

En cualquier caso para determinar el valor de "NUINMI" :

$$\text{NUINMI} = V \text{ TB } Q \quad (71)$$

y la ganancia :

$$G = \frac{\text{NUINMI}}{(V + 1) \text{ TL}} \quad (72)$$

Con este modo de proceder hemos obtenido un método analítico para evaluar teóricamente el "NUINMI" y la ganancia que se produce localmente en un lazo de un programa, ejecutado en un sistema que incorpore el coprocesador para migración vertical dinámica, y para una configuración de M.I. dada. Este método será útil para determinar cual es la configuración de M.I. más apropiada para un sistema, conocido el tipo de programas que ejecuta con más frecuencia. Este estudio ha sido inicialmente realizado para un lazo de 100 instrucciones para luego aplicando un factor de escala obtener resultados para cualquier caso, pero con el método que hemos descrito, los resultados que se obtienen son independientes del tamaño del lazo en términos absolutos; únicamente dependen de la relación entre el tamaño del lazo y el de la M.I., así como de la partición de ésta.

El método de cálculo puede ser resumido en el siguiente algoritmo, teniendo en cuenta que el número de bloques (N) de la partición de M.I. es representado por la variable "NB" :

BEGIN

Q = 0

X = NB - (NP módulo NB)

SI NP \geq 3NB - 1 ENTONCES NUINMI = 0

EN CASO CONTRARIO

BEGIN

SI NP \leq NB ENTONCES NUINMI = V TB NP

EN CASO CONTRARIO

BEGIN

SI $NP \leq 2NB$ ENTONCES

BEGIN

SI $NP = NB + 1$ ENTONCES (* residuos *)

BEGIN

$D = TB - (TL \text{ módulo } TB)$

$Q = D / TB$

END

SI $X \neq 0$ ENTONCES $Q = Q + X$

$Y = (2X + 2NB + 2) / 3$

SI $Y - \text{PARTE ENTERA } (Y) \neq 0$ ENTONCES

$Y = \text{PARTE ENTERA } (Y) + 1$

SI $Y \leq NB$ ENTONCES $Q = Q + NB - Y + 1$

END

EN CASO CONTRARIO (* $2NB < NP < 3NB - 1$ *)

BEGIN

$Y = (2X + 2) / 3$

SI $Y - \text{PARTE ENTERA } (Y) \neq 0$ ENTONCES

$Y = \text{PARTE ENTERA } (Y) + 1$

$Y("1") = X + 1$

SI $Y("1") - Y \leq 0$ ENTONCES $Q = 0$

EN CASO CONTRARIO $Q = Y("1") - Y$

END

$NUINMI = V \cdot TB \cdot Q$

END

$G = NUINMI / ((V + 1) \cdot TL)$

END

END.

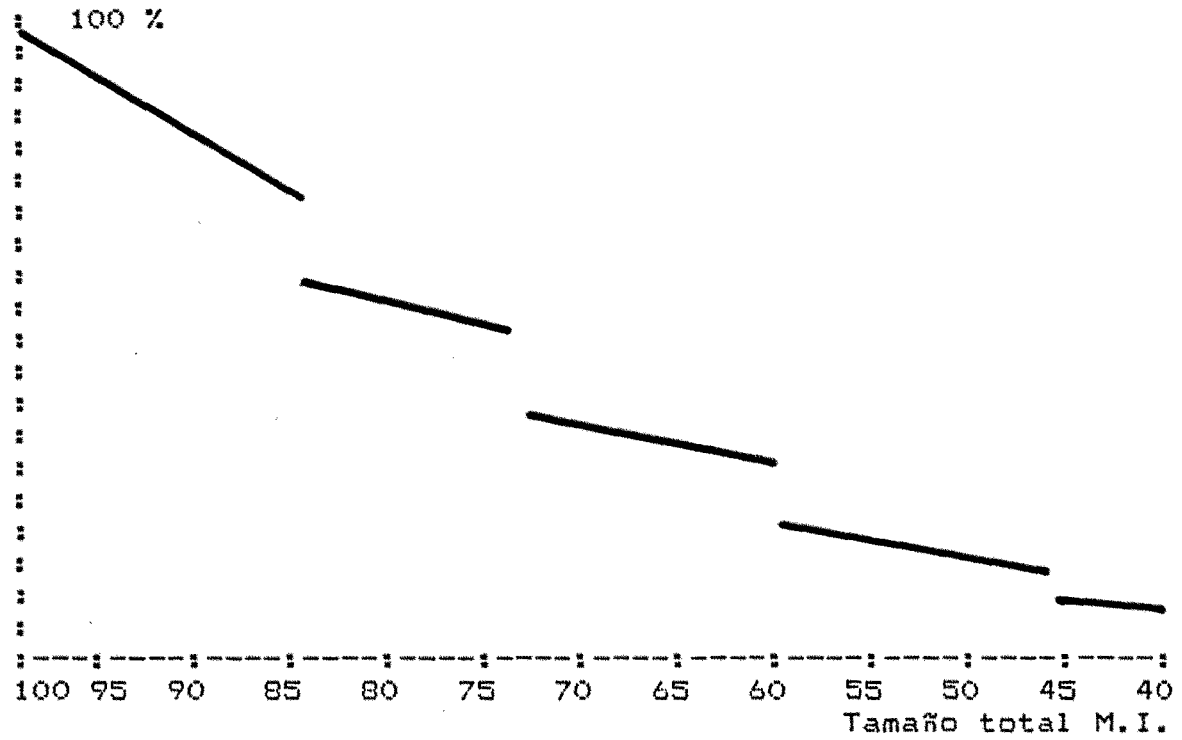
Este algoritmo nos proporciona un modelo analítico de evaluación de la ganancia para un lazo cualquiera. Dada la actuación local del coprocesador, para determinar la ganancia en un programa dado, será preciso aplicar el algoritmo a las diferentes estructuras repetitivas del programa y determinar los "NUINMI" locales. Sumándolos todos se obtiene el "NUINMI" global del programa y dividiendo éste por el número total de instrucciones ejecutadas podemos conocer la ganancia total.

En la gráfica III.5 obtenida del simulador, podemos observar el comportamiento que se ha descrito. Se ha tomado un ejemplo con un tamaño de M.I. inicialmente de 100 posiciones de memoria y se ha ido reduciendo para las diferentes ejecuciones de un lazo de 100 instrucciones con 100 iteraciones (99 vueltas atrás). Véase como cada vez que el lazo disminuye su tamaño en una página se produce un salto brusco en la ganancia, además cada uno de los tramos tiene pendiente distinta puesto que todos ellos apuntan a la abscisa "0". Véase también como en el último tramo la pendiente es mayor (apunta a la abscisa "50"), esto es debido a que la presencia de los residuos produce una ganancia adicional.

GRAFICA III.5

LAZO = 1 <--- 100 (99) M.I. = 6 BLOQUES DE x posiciones

GANANCIA



Se pueden obtener resultados idénticos a la gráfica III.5, si ésta es calculada mediante la aplicación del modelo algorítmico de comportamiento del coprocesador.

CAPITULO IV

UTILIZACION DEL COPROCESADOR : ADAPTACION DE LA ARQUITECTURA EN EL MICROPROCESADOR "R6502"

Para ilustrar la utilización del coprocesador para la adaptación de la arquitectura en tiempo de ejecución, sobre un sistema real, se ha elegido el microprocesador R6502. Este microprocesador está construido sobre un único circuito integrado, de forma que no tenemos acceso a las diferentes partes que lo conforman. Dado que nuestro objetivo se centra en ilustrar la incorporación y funcionamiento del coprocesador, hemos definido una estructura interna para dicho microprocesador, en versión microprogramada y microprogramable, que tenga los mínimos recursos para soportar el repertorio del R6502 con los mismos requerimientos de tiempo que el propio R6502, así como sus particularidades en cuanto a la obtención de resultados parciales e información que circula por los buses (Ro78). Con objeto de emular al R6502 se han desarrollado los microprogramas destinados a su Unidad de Control.

IV.1 DESCRIPCION DEL R6502 VERSION MICROPROGRAMADA

En la parte operativa de la versión microprogramada del R6502 que hemos definido, y que se muestra en la figura IV.1 (He86), se mantienen todos los registros del R6502 y además incluye los siguientes registros :

- G1 y G2 De propósito general, de 8 bits, que son utilizados como registros de almacenamiento temporal, pero que en este trabajo no han sido utilizados.

- Registros de valores constantes (0 y 1), que nos permiten realizar el paso directo a través de la unidad aritmético-lógica (U.A.L.), posibilitando además la función de incremento, el direccionamiento "ZERO PAGE" y los accesos a la memoria pila.

Las características de la UAL son las propias del R6502.

La estructura de la máquina está organizada alrededor de seis buses (DBINT, DBEX, DB1, DB2, ABH y ABL).

DBINT permite las transferencias de información entre la UAL y los registros.

DBEX es el bus de datos que comunica la memoria princi-

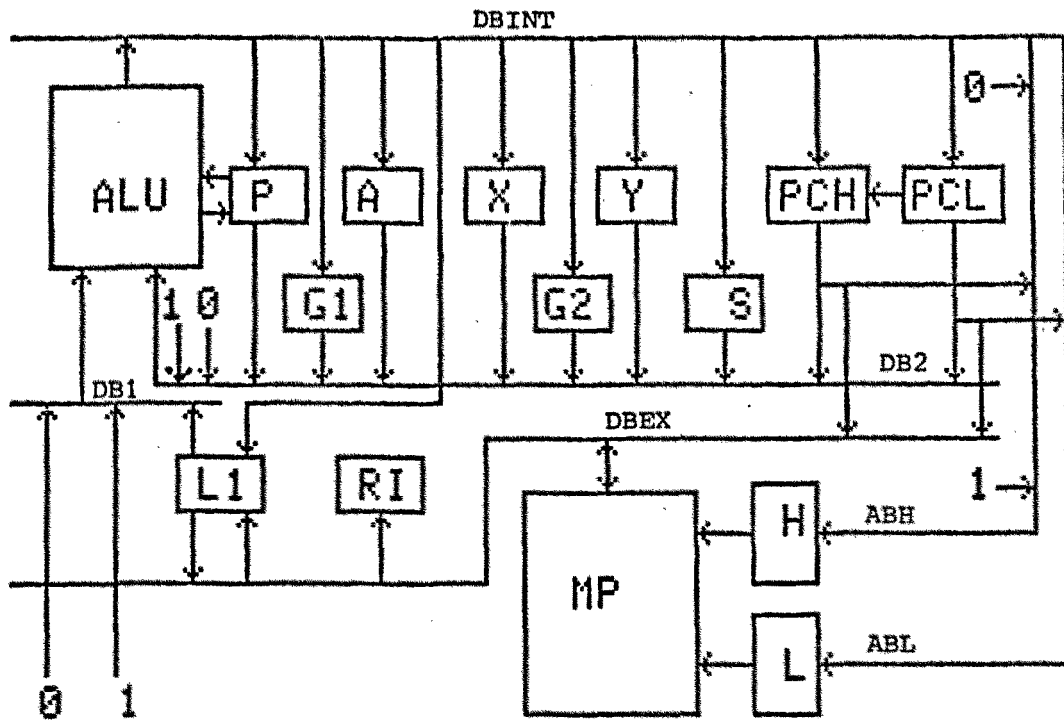


Figura IV.1 Arquitectura Propuesta

pal con el procesador. La conexión de PCH y PCL con la memoria, a través del bus DBEX, tiene la finalidad de poder almacenar sus contenidos en la memoria simultáneamente con otras operaciones.

DB1 es un bus de entrada de datos a la UAL, al que vierten los registros L1 y los de valores constantes 0 y 1.

DB2 es el otro bus de entrada a la UAL, al que vierten los restantes registros operativos.

ABH y ABL configuran el bus de direcciones interno del procesador.

La Unidad de Control del procesador microprogramado (Figura IV.2), ha sido diseñada a partir del secuenciador

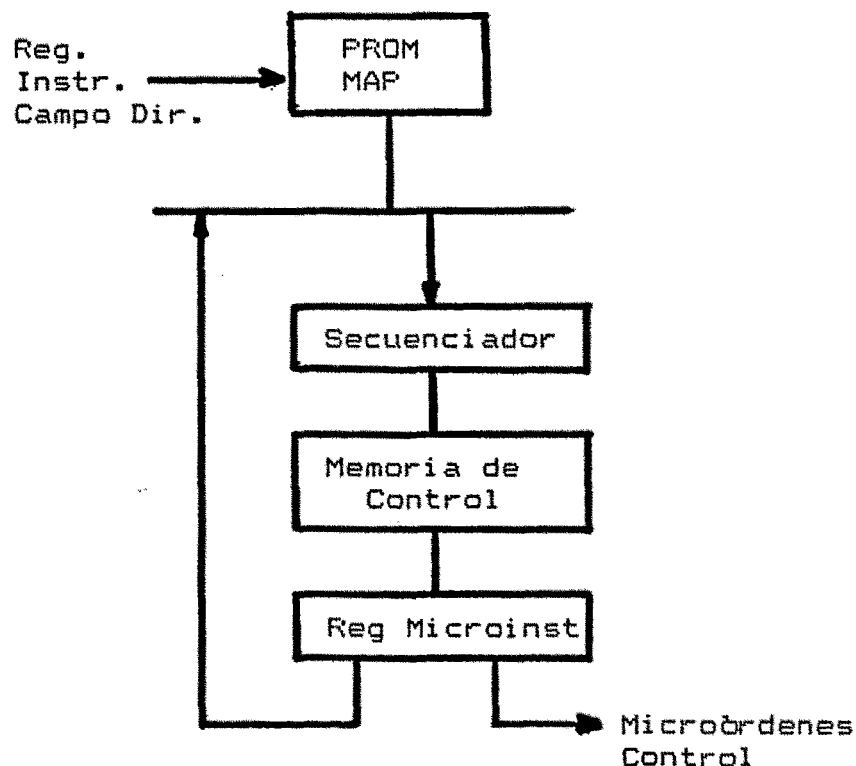


Figura IV.2 Unidad de Control Procesador

AM2910 de AMD, puesto que se dispone de la realización de una Unidad de Control microprogramada y microprogramable (So82). Esta Unidad de Control junto con la parte operativa descrita ha sido utilizada por el grupo de la Universidad de Bari para realizar la emulación completa del mencionado microprocesador en diferentes trabajos (Na84), (No84), (Bi85), (Co185) y (Con85). En esta unidad de control se hace uso del registro "Pipeline" o de microinstrucciones, el cual posibilita el solapamiento de la ejecución de una microinstrucción con la búsqueda de la siguiente.

En el apéndice "D" se muestra el listado de los microprogramas que realizan la interpretación del repertorio del R6502, con los mismos diagramas de tiempo de éste. Para desarrollar estos microprogramas se ha dividido el tiempo de ciclo del R6502 ($T_j = 1$ microsegundo) en cuatro subciclos (T_{j-k}) de 250 microsegundos, en cada uno de los cuales se ejecuta una microinstrucción.

A efectos de nomenclatura describimos la terminología utilizada :

Ta - b	Subciclo "b" del ciclo "a".
<M>	Contenido de la posición de memoria direccionada.
<A> --> B	El contenido del registro "A" debe ser cargado en el registro "B".
INC PC	Incrementar el contador de programa.
JMP	Salto incondicional.
NOP	Tiempo muerto. (no operación).

// Separador utilizado para separar microoperaciones simultáneas.

1 --> A Cargar el valor constante "1" en el registro "A".

<PC> --> H,L Cargar la parte alta de PC en H y la baja en L.

<A/B> --> P Cargar en "P" el contenido del registro "A" o del "B".

MAR Registro de direcciones formado por "H" y "L".

IMAP Dirección proporcionada por el decodificador PROM1. Punto de entrada al microprograma.

ZMAP Proporcionada por PROM1. (2o. nivel decodificación). Parte operativa.

3MAP Dirección de entrada a la parte operativa del microprograma para su ejecución en versión "J+1/2". Proporcionada por PROM2. Se almacena en el primer campo de la interconexión.

Existe un microprograma general de búsqueda de la instrucción, común a todas ellas y cuya descripción damos a continuación de forma detallada. Su función consiste en cargar en el registro de instrucciones el código de operación de la instrucción a interpretar, preparar el registro de direcciones (MAR) con la dirección de la posición de la memoria a la que tengamos que acceder a continuación, y en función de la decodificación del código de operación, bifurcar al punto de entrada al microprograma (IMAP) de interpretación, proporcionado por el decodificador FROM 1.

```

TO - 1  START b : <PC> --> H.L (MAR) // Parte operativa
                                     // JMP PEPE           instrucción
                                                         anterior
PEPE TO - 2  NOP
TO - 3  INC PC

```

T0 - 4 <<MAR>> --> Reg. Ins. (Código operación)
T1 - 1 <PC> --> H,L // Decodificación en PROM 1
T1 - 2 JMP IMAP (salto al microprograma de interpretación. IMAP procede de PROM 1).

IMAP T1 - 3 Primera microinstrucción del microprograma

En el primer subciclo (T0 - 1) debemos distinguir la posibilidad del "RESET" del sistema mediante el conjunto de microórdenes resumidas por "START", del caso que corresponde al ciclo normal de búsqueda de una instrucción durante la ejecución de un programa, en el que deberemos inicializar el registro de direcciones con el contenido en curso del contador de programa, para la lectura del código de la instrucción desde la memoria principal. En realidad esta microinstrucción corresponde a la última del microprograma de interpretación de la instrucción anterior, por ello simultáneamente debe ejecutarse la parte operativa de la misma. El salto JMP PEPE nos lleva al microprograma general de búsqueda común a todas las instrucciones.

En el tercer subciclo mientras se produce la espera para la obtención del código de operación, de la memoria, se incrementa el contador de programa puesto que con independencia del tipo de instrucción de que se trate, deberemos leer la posición de memoria siguiente, ya sea un dato de la corriente, ya sea el código de la siguiente instrucción. Una vez obtenido el código se almacena en el registro de instrucciones en T0 - 4 y en T1 - 1 se decodifica a través de la memoria "PROM 1", mientras se actualiza el MAR para la lectura de la siguiente posición de memoria. En T1 - 2 se

realiza el salto al microprograma de interpretaci3n propiamente dicho mediante la direcci3n proporcionada por el decodificador.

IV.2 INCORPORACION DEL COPROCESADOR

Dada la actuaci3n del coprocesador, descrita en los capitulos I y II, la Unidad de Control de este puede ser eliminada aprovechando el secuenciamiento del propio procesador, reduciéndose la estructura del coprocesador a dos m3dulos b3sicos : Memoria de Interconexi3n y Unidad de Gest3n de la Memoria de Interconexi3n. La M.I. se adopta con un tiempo de acceso igual al de la memoria de control es decir de 250 nanosegundos.

Con motivo de eliminar la Unidad de Control del Coprocesador y controlar la b3squeda y generaci3n de las interconexiones, simultáneamente a la ejecuci3n de las microinstrucciones originales, la Unidad de Control del procesador (R6502) microprogramado deber3 sufrir ciertas modificaciones, tales como el aumento de longitud de palabra de la Memoria de Control, para ubicar las micro3rdenes que controlen la M.I. y la U.G.M.I..

A pesar de este cambio de longitud de palabra, la memoria de control continuar3 almacenando los microprogramas originales que interpretan el repertorio del R6502. En el ap3ndice "E" se presentan las microoperaciones que deber3n

realizarse en paralelo con cada microinstrucción original para realizar el control del coprocesador y construir así el nuevo intérprete "J+1/2". Los dirección de los parámetros que se obtienen de la instrucción y deben ser almacenados en la M.I., dado que durante la ejecución normal del microprograma se almacenan en "H" y en "L" (MAR), se aprovecha su presencia en estos registros para almacenarlos en M.I. y completar la interconexión. Estos cambios a nivel del microprograma de búsqueda, común para todas las instrucciones del repertorio, sólo afectan al subciclo TO - 1 (2MAP) al que tenemos que añadir :

```

2MAP      TO - 1  <Campo Pag. M.I. Leida> CMP <HI> // C(UAL)
              = 0 (para A-2-5) // SI comparac. = igual
              ENTONCES : ( <HB> --> H // <LB> --> L //
              INC PC // JMP 3MAP ) EN CASO CONTRARIO : (
              <PC> --> H,L // JMP TO - 2 ).

```

Se realiza la comparación del contenido del campo de la posición de M.I. leida, correspondiente a la página de la memoria principal a la que hace referencia la interconexión, con el contenido del registro de página "HI". Si la comparación resulta positiva se carga el registro MAR (H y L) con los contenidos correspondientes a los campos "HB" y "LB" de la interconexión que constituyen la parte alta y baja respectivamente de la dirección que contiene la instrucción, y que servirá para calcular la dirección del operando, sino se trata de direccionamiento directo, o el propio operando si se tratase de direccionamiento inmediato. Seguidamente se incrementa el contador de programa y se realiza un salto al microprograma que ejecuta la versión

"J+1/2" de la instrucción en curso.

En este caso hemos optado por escribir el microprograma que ejecuta la versión "J+1/2" de la instrucción como un microprograma independiente del original, pero esto podría ser substituido por un salto a la parte operativa del microprograma original de modo que la actuación de los diferentes campos de las microinstrucciones sobre los recursos de la máquina se controlasen en función de que esta microinstrucción fuera ejecutada por la versión normal o la "J+1/2", evitando con ello la duplicación de microinstrucciones.

Existen tres grupos de instrucciones del repertorio del R6502 : A-2-1, A-2-5 y A-6-1, en las que la información que debe almacenarse en los campos "HB" y "LB" de la interconexión, se obtienen al final de todo del microprograma, con lo que no se dispone de tiempo, dentro del mismo microprograma de incluir esta información en la interconexión durante su creación en M.I.. Es por tanto necesario crear un mecanismo para que en estos dos casos, durante la ejecución del microprograma de la siguiente instrucción, cuando exista un hueco en el que no se haga referencia a la M.I. pueda escribirse esta información en la interconexión. Esto supone almacenar temporalmente esta información en dos registros auxiliares que denominamos "HBI" y "LBI".

Normalmente los contenidos de "PCH" y "PCL" se

almacenan en en los registros "HI" y "LI" respectivamente, con la finalidad de que durante el microprograma pueda cambiar el contenido del contador de programa y esto no afecte a la M.I. ni sus contenidos. Para que pueda realizarse la escritura retardada en M.I., durante la ejecución del microprograma siguiente, es necesario guardar la dirección de la interconexión anterior en "HI2" y en "LI2" y para que no haya ningún conflicto con el posible resultado de la siguiente lectura de M.I., debemos cargarlo en un registro "BMI" (figura IV.3).

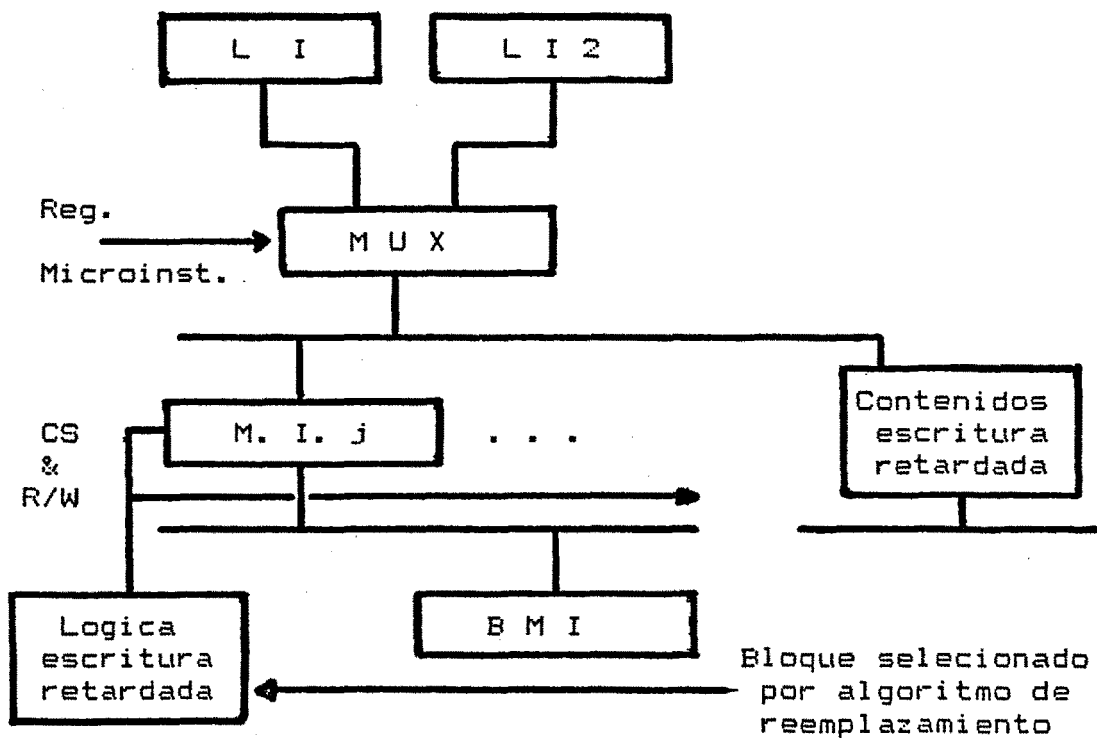


Figura IV.3 Escritura retardada

IV.3 EJEMPLO DESARROLLADO SOBRE LA MAQUINA PROPUESTA

Se ha desarrollado un ejemplo de programa ejecutado en este sistema basado en el R6502 microprogramado y microprogramable tomando un caso de ordenación por el método de la burbuja :

```

                ($0200) LDA  NUMEROS
                   STA  CUENTA
                   DEC  CUENTA
                   LDA  #$01
                   STA  INDICE
SALTO5 ($020E) LDA  #$00 <-----'
                   STA  FLAG
                   TAX
SALTO3 ($0214) LDA  TABLA+1 , X <-----'
                   CMP  TABLA , X
                   ($021A) BCS  SALTO1 -----'
                   LDY  TABLA , X
                   STA  TABLA , X
                   TYA
                   STA  TABLA+1 , X
                   LDA  #$01
                   STA  FLAG
SALTO1 ($022B) SEC  <-----'
                   LDA  CUENTA
                   SBC  INDICE
                   STA  NUMEROS+1
                   CPX  NUMEROS+1
                   ($0238) BEQ  SALTO2 -----'
                   INX
                   ($023B) JMP  SALTO3 -----'
SALTO2 ($023E) LDA  FLAG <-----'
                   CMP  #$00
                   ($0243) BEQ  SALTO4 -----'
                   LDA  CUENTA
                   CMP  INDICE
                   ($024B) BEQ  SALTO4 ----->
                   INC  INDICE
                   ($0250) JMP  SALTO5 -----'
SALTO4 ($0253) BRK  <-----'

```

Si tomamos una M.I. de 4 bloques de 10 bytes cada uno y con un conjunto de datos de entrada de cinco números ordenados inicialmente al revés, el resultado que se obtiene

nos da :

2804 microsegundos para la ordenación en
 ejecución normal

1303 microsegundos para la ordenación en
 ejecución con el coprocesador en operación

Es decir que el tiempo de ejecución se reduce al 46 % del original por el hecho de haber introducido el coprocesador en el sistema, teniendo en cuenta que el tamaño total de M.I. es de 40 bytes, que el tamaño del programa es de 84 bytes y 34 instrucciones, y que el tamaño de los dos lazos es de 69 bytes (28 instrucciones) para el mayor y de 42 bytes (17 instrucciones) para el menor. El tamaño de la M.I. es menor que el tamaño del menor de los lazos.

Es importante destacar que en el ejemplo que presentamos sobre el R6502, una parte importante de la ganancia se produce por el hecho de que el formato de las instrucciones del repertorio de dicho microprocesador son de 1, 2 o 3 bytes, mientras que la interconexión que genera cada una es de una sola palabra. El resultado es que al irse almacenando las interconexiones en la M.I. durante la ejecución del programa, se producen huecos entre interconexiones correspondientes a instrucciones consecutivas, por lo que en posteriores asignaciones de un bloque de M.I. un número nada despreciable de interconexiones relativas a la nueva asignación del bloque, pasa a ocupar los huecos, respetando una parte importante de la información perteneciente a la asignación antigua. En el caso de que el bloque sea asignado de

nuevo a la página anterior, encontraremos todas las interconexiones que no han sido destruidas en la reasignación.

Puesto que hemos configurado la M.I. como cuatro bloques (A, B, C y D) de diez posiciones (bytes) cada uno, las páginas de la memoria principal por las que transcurrirá el programa son :

Pag 0 : \$0200 a \$0209

Pag 1 : \$020A a \$0213

Pag 2 : \$0214 a \$021D

Pag 3 : \$021E a \$0227

Pag 4 : \$0228 a \$0231

Pag 5 : \$0232 a \$023B

Pag 6 : \$023C a \$0245

Pag 7 : \$0246 a \$024F

Pag 8 : \$0250 a \$0259

Si indicamos con "x" la nueva creación de una interconexión y con "e" cuando nos encontramos con ella ya creada, omitiendo las dos cifras mas significativas de la dirección (\$02), la evolución de los contenidos de los bloques será :

BLOQUE "A"

Durante la ejecución se asigna a las páginas :

0, 4, 8, 1, 4, 8, y 3. Sus contenidos :

Pos M.I.	Posiciones de la memoria principal
\$00	! (00)x (28)x eee (50)x (28)xee (50)xe
\$01	!
\$02	!
\$03	! (03)x (2B)x eee eee eee
\$04	!
\$05	! (23)x ee
\$06	! (06)x (10)x e
\$07	! (2F)x eee eee eee
\$08	!
\$09	! (09)x (13)x ee

Como puede verse en la posición \$00 al producirse las nuevas asignaciones del bloque a las siguientes páginas, se destruye su contenido anterior (el 28 sustituye al 50 y viceversa), pero por ejemplo en \$03, \$05 y \$07 que contienen interconexiones de la página "4" mantienen su contenido a pesar de la reasignación del bloque y son recuperados cuando el bloque es de nuevo asignado a la página "4". Algo parecido ocurre con los demás bloques :

BLOQUE "B"

Durante la ejecución se asigna a las páginas :

1 y 5. Sus contenidos :

Pos M.I.	Posiciones de la memoria principal
\$00	! (32)x eee eee eee
\$01	! (0B)x
\$02	!
\$03	! (35)x eee eee eee
\$04	! (0E)x eee
\$05	!
\$06	! (10)x (38)x eee eee eee
\$07	!
\$08	! (3A)x ee ee e
\$09	! (13)x (3B)x ee ee e

BLOQUE "C"

Durante la ejecución se asigna a las páginas :

2 y 6. Sus contenidos :

Pos Posiciones de la memoria principal
M.I.

```
-----  
$00 ! (14)x eee eee eee  
$01 !  
$02 ! (3E)x e e e  
$03 ! (17)x eee eee eee  
$04 !  
$05 ! (41)x eee  
$06 ! (1A)x eee eee eee  
$07 ! (43)x eee  
$08 ! (1C)x eee eee eee  
$09 ! (45)x eee
```

BLOQUE "D"

Durante la ejecución se asigna a las páginas :

3, 7, 3, 7 y 4. Sus contenidos :

Pos Posiciones de la memoria principal
M.I.

```
-----  
$00 ! (28)x ee  
$01 ! (1F)x eee eee eee  
$02 ! (48)x eee  
$03 !  
$04 ! (22)x eee eee eee  
$05 ! (23)x eee (4B)x (23)x ee (4B)x ee FIN  
$06 !  
$07 ! (4D)x ee  
$08 ! (26)x eee eee eee  
$09 !
```

Se ha ejecutado el mismo ejemplo con una configuración de M.I. de cuatro bloques de 18 posiciones (bytes) cada uno obteniéndose una reducción del tiempo de ejecución a un 44% del original, es decir que aumentando considerablemente el tamaño de la M.I. no se consigue un aumento apreciable de la ganancia temporal.

CONCLUSIONES Y LINEAS ABIERTAS

- Se ha desarrollado un método de migración vertical dinámica basado en la incorporación de un coprocesador que realice la tarea de migración en tiempo de ejecución de un programa.

- Este método reúne las características de ser totalmente transparente al usuario, de seguir constantemente el perfil de ejecución de un programa (adaptivo), y de ser igualmente eficiente en medios estacionarios, quasi-estacionarios y no estacionarios. La migración se efectúa en tiempo de ejecución y en paralelo con ella, sin añadir ningún tiempo extra al de ejecución normal de un programa.

- El método propuesto, a diferencia de los métodos clásicos; no precisa de análisis estadísticos del flujo de ejecución de los programas en función de los datos de entrada, por ello no requiere el uso de técnicas de monitorización firmware, ni de ningún estudio previo a la ejecución del programa para determinar los límites de los lazos y su microprogramabilidad.

- La capacidad dinámica para aplicar localmente los

recursos del coprocesador hacen que la estrategia de adaptación propuesta sea especialmente útil en aquellos sistemas que presentan una movilidad elevada de programas (ejemplo medios multiprogramados), donde los métodos clásicos no ofrecen elevados rendimientos debido a sus estrategias globales de adaptación.

- Dado un programa escrito en el lenguaje de nivel "K", interpretado en el nivel "J", se crea un nuevo lenguaje intermedio, que hemos denominado "J+1/2", que interconecta los niveles "K" y "J". Al ejecutar una sentencia escrita en el lenguaje de nivel "K", se genera en tiempo de ejecución una interconexión en el nivel "J+1/2" (sentencia de dicho lenguaje), la cual es interpretada de forma más rápida en el mismo nivel "J", mediante un nuevo intérprete. Cuando una sentencia del lenguaje de nivel "K" ha sido ejecutada y con ello creada su interconexión, en posteriores ejecuciones de la misma (por iteración), se ejecutará su correspondiente sentencia en "J+1/2" generando con ello un aumento de velocidad en la ejecución del programa.

- Se ha desarrollado un programa simulador del comportamiento de un sistema incorporando el coprocesador, en dos versiones, una determinística y otra probabilística, que permiten el estudio del comportamiento del sistema para diferentes configuraciones de parámetros y diferentes programas. Analizando los resultados proporcionados por el simulador ha sido posible elaborar un modelo algorítmico que

nos permite calcular la ganancia esperada en un sistema, para un programa dado, conocida la configuración de parámetros que posee.

Hemos creído interesante comentar las posibles líneas que este trabajo deja abiertas para ser estudiadas en un futuro.

A- La aplicación del coprocesador a la ejecución interpretada de lenguajes de alto nivel.

B- Ampliar el modelo algorítmico de comportamiento del coprocesador para que pueda incluir las estructuras cíclicas de longitud variable.

C- Analizar diferentes alternativas para aumentar las prestaciones del sistema propuesto :

- La inclusión en el coprocesador de la capacidad de tratamiento específico de las subrutinas, dado que en el esquema clásico de migración éstas constituirían clases de equivalencia.

- Funcionamiento asíncrono del coprocesador con respecto al procesador que permita el solapamiento de fases no idénticas de ambos y que permitan que la lectura de la M.I. pueda realizarse con anterioridad a la memoria principal.

- La lectura en memoria principal de las sentencias que presumiblemente han de seguir a la que se halla en curso y realizar su migración por adelantado. Con ello aumentaríamos la probabilidad de que al ejecutar una sentencia ya encontremos su versión "J+1/2".

- Utilizar el lenguaje intermedio "J+1/2" generado sobre la base de paralelizar la ejecución de varias sentencias del lenguaje original (paralelismo migrado) en el sentido de simultanear aquellas partes de diferentes sentencias que no entren en conflicto. Para ello pensamos que sería necesario definir la arquitectura del procesador (estructura más repertorio) que permita aprovechar al máximo el paralelismo migrado. En el caso de realizarlo a nivel de lenguaje máquina y microprogramación, deberíamos analizar la compatibilidad de los diferentes campos de las microinstrucciones.

APENDICE A
LISTADO SIMULADOR DETERMINISTICO
DETERPRIM

```
(*G*)  
PROGRAM DETERPRIM;  
USES  
  APPLESTUF;  
TYPE  
  ALGORITMO = 1..3;  
  SELECCION = 0..6;  
VAR  
  ALGO : ALGORITMO;  
  SEL : SELECCION;  
  LISTA : ARRAY [1..1000] OF RECORD  
  SALTO,VUELTAS,UVU: INTEGER  
  END;  
  ESTA,EXBO : BOOLEAN;  
  REGHISTORIA: ARRAY [1..99,1..8] OF INTEGER;  
  RELRU: ARRAY [1..99] OF INTEGER;  
  MEMI : ARRAY [1..1000] OF INTEGER;  
  BLOQUE,BRANC,TAMBUMAX,NUMINST,NUINMI,TAMBO,  
  NUMBO,ORG,DEST,VUEL,LOGLIS,POSESCRI,  
  PAG,NUMREG,POBO,POPRO,TAMTOMI,  
  PAR, X,S,V : INTEGER;  
  CH, ALTO:CHAR;  
  GANAN:REAL;  
PROCEDURE LONGLISTA;(*PIDE LA LONGITUD DEL PROGRAMA*)  
  BEGIN  
    WRITE (' J LONGITUD DEL PROGRAMA ? = ');  
    READLN (LOGLIS);  
    WRITELN(' ');  
  END;  
PROCEDURE MEMINTER;  
  BEGIN  
    WRITE (' J BLOQUES DE MEM. DE INTERC. ? ');  
    READLN(NUMBO);  
    WRITE (' J TAMA/O EN BYTES DE CADA BLOQUE ? ');  
    READLN(TAMBO);  
  END;  
PROCEDURE REGISTROS;  
  BEGIN  
    WRITE (' J CUANTOS REGISTROS DE HISTORIA ? ');  
    READLN(NUMREG);  
  END;  
PROCEDURE INICIALISTA;(*PONE TODO EL PROGRAMA INICIALMENTE A CERO*)  
  VAR A : INTEGER;  
  BEGIN  
    FOR A := 1 TO LOGLIS DO  
      BEGIN
```



```

        WITH LISTA[A] DO
        BEGIN
            SALTO := 0;
            VUELTAS := 0;
            CUVU := 0;
        END;
    END;
END;
PROCEDURE INIMEMI;
VAR A : INTEGER;
BEGIN
    TAMTOMI:=TAMBO*NUMBO;
    FOR A:=1 TO TAMTOMI DO
    BEGIN
        MEMI[A]:=0;
    END;
END;
PROCEDURE INIREG;
VAR R,B : INTEGER;
BEGIN
    FOR R:=1 TO NUMREG DO
    BEGIN
        FOR B:=1 TO NUMBO DO
        BEGIN
            REGHISTORI[B,R]:=-B;
        END;
    END;
END;
PROCEDURE INILRU;
VAR
    A : INTEGER;
BEGIN
    FOR A := 1 TO NUMBO DO
    BEGIN
        RELRU[A] := A;
    END;
END;
PROCEDURE ACTUALRU;
VAR
    A , SALVA, LUGAR : INTEGER;
BEGIN
    FOR A := 1 TO NUMBO DO
    BEGIN
        IF RELRU[A] = BLOQUE THEN
        BEGIN
            LUGAR := A;
            SALVA := RELRU[A];
        END;
    END;
    FOR A := LUGAR DOWNTO 2 DO
    BEGIN
        RELRU[A] := RELRU[A-1];
    END;
END;

```

```

RELRU[1] := SALVA;
END;
PROCEDURE SELAL;
BEGIN
  WRITELN('1 MAPPING DIRECTO');
  WRITELN('2 MAX. DISTANCIA');
  WRITELN('3 L. R. U. ');
  WRITE('SELECCION = ');
  READLN(ALGO);
END;
PROCEDURE ENTRARVAL;
BEGIN
  WRITELN(' ');
  WRITELN('INDIQUE EL NUMERO DE VECES');
  WRITE('QUE DEBE EJECUTARSE UN BUCLE');
  WRITELN(', SUS LIMITES Y EL NUMERO');
  WRITELN('DE ACTIVACIONES');
  WRITELN(' ');
  REPEAT
    WRITE('DESDE DONDE ? ');
    READLN(ORG);
    WRITE('HASTA DONDE ? ');
    READLN (DEST);
    LISTA[ORG].SALTO:=DEST;
    WRITE ('CUANTAS VUELTAS ? ');
    READLN(VUEL);
    LISTA[ORG].VUELTAS:=VUEL;
    LISTA[ORG].CUVU := VUEL;
    READLN(ALTO);
  UNTIL ALTO='0';
END;
PROCEDURE BUSQUEDA;
VAR
  A,B: INTEGER;
BEGIN
  ESTA := FALSE;
  IF POPRO MOD TAMBO = 0 THEN POBO := TAMBO
  ELSE POBO := POPRO MOD TAMBO;
  FOR A := 1 TO NUMBO DO
    BEGIN
      B := POBO + TAMBO * (A-1);
      IF MEMI[B] = POPRO THEN ESTA:= TRUE;
    END;
  END;
END;
PROCEDURE MAPDIREC;
BEGIN
  IF PAG MOD NUMBO = 0 THEN BLOQUE := NUMBO

```

```

        ELSE BLOQUE := PAG MOD NUMBO;
    END;
PROCEDURE LRU;
    BEGIN
        BLOQUE := RELRUC[ NUMBO ];
    END;
PROCEDURE HISTORIA;
    VAR
        B,R: INTEGER;
    BEGIN
        EXBO := FALSE;
        FOR R := 1 TO NUMREG DO
            BEGIN
                FOR B := 1 TO NUMBO DO
                    BEGIN
                        PAG := (POPRO-1) DIV TAMBO + 1;
                        IF REGHISTORIA[B,R] = PAG THEN
                            BEGIN
                                EXBO := TRUE;
                                BLOQUE := B;
                            END;
                        END;
                    END;
                END;
            END;
        END;
PROCEDURE DISTANCIA;
    VAR
        A,B,C,PRIMER,SEGON : INTEGER;
    BEGIN
        C := 0;
        PRIMER:=1;
        SEGON:=0;
        FOR B := 1 TO NUMBO DO
            BEGIN
                A := ABS(PAG - REGHISTORIA[B,1]);
                IF A > C THEN
                    BEGIN
                        C := A;
                        PRIMER:=B;
                        SEGON:=0;
                    END
                ELSE IF A=C THEN SEGON := B;
            END;
        END;
        IF SEGON = 0 THEN BLOQUE := PRIMER
        ELSE IF REGHISTORIA[PRIMER,1]<REGHISTORIA[SEGON,1] THEN BLOQUE := PRIMER
        ELSE BLOQUE := SEGON;
    END;
PROCEDURE ACTUALHIS;

```

```

VAR
  R: INTEGER;
BEGIN
  FOR R := NUMREG DOWNT0 1 DO
    BEGIN
      IF R=1 THEN REGHISTORIA[BLOQUE,R]:= PAG
      ELSE REGHISTORIA[BLOQUE,R] := REGHISTORIA[BLOQUE,R-1];
    END;
  END;
PROCEDURE LAZOS;
VAR
  A : INTEGER;
BEGIN
  WRITELN(CHR(12));
  CASE ALGO OF
    1 : WRITELN('MAPPING DIRECTO');
    2 : WRITELN('ALGORITMO DISTANCIA');
    3 : WRITELN('L. R. U. ');
  END;
  WRITE('HISTOR = ',NUMREG);
  WRITELN(' BLOQUES = ',NUMBO,' DE ',TAMBO);
  WRITE('PROGRAMA DE ',LONGLIS);
  WRITELN(' CON LOS LAZOS :');
  FOR A:=1 TO LONGLIS DO
    BEGIN
      IF LISTA[A].SALTO <> 0 THEN
        BEGIN
          WRITELN(LISTA[A].SALTO:4,' <--- ',A:4,
            ' (',LISTA[A].VUELTAS:4,')');
        END;
    END;
  REPEAT
  UNTIL KEYPRESS;
END;
PROCEDURE IMPRIMIR;
VAR
  Q : FILE OF CHAR;
  A : INTEGER;
BEGIN
  REWRITE(Q,'PRINTER:PP');
  WRITE(Q,'DETALLMI ');
  CASE ALGO OF
    1 : WRITELN(Q,'MAPPING DIRECTO');
    2 : WRITELN(Q,'ALGORITMO DISTANCIA');
    3 : WRITELN(Q,'L. R. U. ');
  END;
  WRITE(Q,'NORMAL = ',NUMINST);

```

```

WRITE(Q,' M I = ',NUINMI);
WRITELN(Q,' GANAN = ',GANAN,'%');
WRITE(Q,'HISTOR = ',NUMREG);
WRITELN(Q,' BLOQUES = ',NUMBO,' DE ',TAMBO);
WRITE(Q,'PROGRAMA DE ',LONGLIS);
WRITELN(Q,' CON LOS LAZOS :');
FOR A:=1 TO LONGLIS DO
  BEGIN
    IF LISTA[A].SALTO <> 0 THEN
      BEGIN
        WRITELN(Q,LISTA[A].SALTO:4,' <--- ',A:4,
          ' (',LISTA[A].VUELTAS:4,')');
      END;
    END;
  WRITELN(Q,' ');
  WRITELN(Q,' ');
  CLOSE(Q,LOCK);
END;
PROCEDURE RESULTADOS;
VAR
  A : INTEGER;
BEGIN
  WRITELN(CHR(12));
  WRITE('DETALLMI : ');
  CASE ALGO OF
    1 : WRITELN('MAPPING DIRECTO');
    2 : WRITELN('ALGORITMO DISTANCIA');
    3 : WRITELN('L. R. U. ');
  END;
  WRITE('NORMAL = ',NUMINST,' ');
  WRITELN('M I = ',NUINMI);
  WRITELN('GANANCIA = ',GANAN,'%');
  WRITE(' HISTOR = ',NUMREG,' ');
  WRITELN('BLOQUES = ',NUMBO,' DE ',TAMBO);
  WRITE('PROGRAMA DE ',LONGLIS);
  WRITELN(' CON LOS LAZOS :');
  WRITELN(' ');
  FOR A:=1 TO LONGLIS DO
    BEGIN
      IF LISTA[A].SALTO <> 0 THEN
        BEGIN
          WRITELN(LISTA[A].SALTO:4,' <--- ',A:4,
            ' (',LISTA[A].VUELTAS:4,')');
        END;
      END;
    WRITELN(' ');
    WRITELN(' ');
    WRITE('IMPRIMIR = P ');
    READLN(ALTO);
    IF ALTO = 'P' THEN IMPRIMIR;
  END;
END;

```

```

PROCEDURE MENU;
BEGIN
  WHILE SEL < 6 DO
  BEGIN
    WRITELN(CHR(12));
    WRITELN(' ');
    WRITELN('0  VISUALIZACION ESTADO');
    WRITELN(' ');
    WRITELN('1  CAMBIAR BLOQUES M.I. ');
    WRITELN(' ');
    WRITELN('2  NUM. REGIS. HIST. ');
    WRITELN(' ');
    WRITELN('3  SALTOS.           ');
    WRITELN(' ');
    WRITELN('4  RESET.           ');
    WRITELN(' ');
    WRITELN('5  SELECCION ALGORITMO. ');
    WRITELN(' ');
    WRITELN('6  EJECUTAR           ');
    WRITELN(' ');
    WRITELN('7  TERMINAR.         ');
    WRITELN(' ');
    WRITE('                SELECCION: ');
    READLN(SEL);
    CASE SEL OF
      0 : LAZOS;
      1 : BEGIN
          MEMINTER;
          INIMEMI;
        END;
      2 : BEGIN
          REGISTROS;
          INIREG;
        END;
      3 : ENTRARVAL;
      4 : BEGIN
          LONGLISTA;
          MEMINTER;
          REGISTROS;
          INICIALISTA;
          INIMEMI;
          INIREG;
          ENTRARVAL;
          SELAL;
        END;
      5 : SELAL;
      6,7:;
    END;
  END;

```

```
END;  
END;
```

```
BEGIN
```

```
SEL := 4;  
MENU;
```

```
WHILE SEL < 7 DO
```

```
  BEGIN
```

```
    INIMEMI;  
    INIREG;  
    IF ALGO = 3 THEN INILRU;  
    POPRO:=1;  
    NUMINST:=0;  
    NUINMI:=0;
```

```
  WHILE POPRO <= LONGLIS DO
```

```
    BEGIN
```

```
      NUMINST := NUMINST +1;
```

```
      BUSQUEDA;
```

```
      IF ESTA = TRUE THEN NUINMI := NUINMI + 1
```

```
      ELSE
```

```
        BEGIN
```

```
          HISTORIA;
```

```
          IF EXBO = FALSE THEN
```

```
            BEGIN
```

```
              CASE ALGO OF
```

```
                1 : MAPDIREC;
```

```
                2 : DISTANCIA;
```

```
                3 : LRU;
```

```
            END;
```

```
          END;
```

```
          (* ESCRIBIR *);
```

```
          POSESCRI := POBO + TAMBO * (BLOQUE -1);
```

```
          MEMI[POSESCRI]:= POPRO;
```

```
          (*SOLO HARIA FALTA ESCRIBIR LA PARTE ALTA DE POPRO*);
```

```
          ACTUALHIS;
```

```
        END;
```

```
      IF (ALGO = 3) AND (NUMBO > 1) THEN ACTUALRU;
```

```
      IF (LISTA[POPRO].SALTO=0) OR (LISTA[POPRO].CUVU=0) THEN
```

```
        BEGIN
```

```
          LISTA[POPRO].CUVU := LISTA[POPRO].VUELTAS;
```

```
          POPRO:=POPRO + 1;
```

```
        END
```

```
      ELSE
```

```
BEGIN
  IF LISTA[POPPOJ].VUELTAS = 32000 THEN
    POPRO := LISTA[POPPOJ].SALTO
  ELSE
    BEGIN
      LISTA[POPPOJ].CUVU := LISTA[POPPOJ].CUVU - 1;
      POPRO := LISTA[POPPOJ].SALTO;
    END;
  END;
END;

GANAN := (NUINMI/NUMINST)*100;
RESULTADOS;
SEL := 4;
MENU;
END;

END.
```


APENDICE B
LISTADO SIMULADOR SISTEMATICO

DETSISTMI

```
PROGRAM DETSISTMI;
USES
  APPLESTUF;
TYPE
  ALGORITMO = 1..2;
  SELECCION = 0..6;
VAR
  ALGO : ALGORITMO;
  SEL : SELECCION;
  LISTA : ARRAY [1..1000] OF RECORD
  SALTO, VUELTAS, CUVU: INTEGER
  END;
  ESTA, EXBO : BOOLEAN;
  REGHISTORIA: ARRAY [1..32, 1..8] OF INTEGER;
  MEMI : ARRAY [1..1000] OF INTEGER;
  BLOQUE, BRANC, TAMBUMAX, NUMINST, TAMBO,
  NUMBO, ORG, DEST, VUEL, LONGLIS, POSESCRI,
  PAG, NUMREG, POBO, POPRO, TAMTOMI: INTEGER;
  ALTO: CHAR;
  GANAN: ARRAY [1..32] OF REAL;
  NUINMI : ARRAY [1..32] OF INTEGER;
PROCEDURE LONGLISTA; (*PIDE LA LONGITUD DEL PROGRAMA*)
  BEGIN
    WRITE (' ] LONGITUD DEL PROGRAMA ? = ');
    READLN (LONGLIS);
    WRITELN(' ');
  END;
PROCEDURE MEMINTER;
  BEGIN
    WRITE('TAMA/O TOTAL DE M.I. ? ');
    READLN(TAMTOMI);
  END;
PROCEDURE REGISTROS;
  BEGIN
    WRITE('] CUANTOS REGISTROS DE HISTORIA ? ');
    READLN(NUMREG);
  END;
PROCEDURE INICIALISTA; (*PONE TODO EL PROGRAMA INICIALMENTE A CERO*)
  VAR A : INTEGER;
  BEGIN
    FOR A := 1 TO LONGLIS DO
      BEGIN
        WITH LISTA[A] DO
          BEGIN
            SALTO := 0;
            VUELTAS := 0;
            CUVU := 0;
          END;
        END;
      END;
  END;
```

```

        END;
    END;
END;
PROCEDURE INIMEMI;
VAR A : INTEGER;
BEGIN
    FOR A:=1 TO TAMTOMI DO
        BEGIN
            MEMICAJ:=0;
        END;
    END;
END;
PROCEDURE INIREG;
VAR R,B : INTEGER;
BEGIN
    FOR R:=1 TO NUMREG DO
        BEGIN
            FOR B:=1 TO NUMBO DO
                BEGIN
                    REGHISTORIC[B,R]:=-B;
                END;
            END;
        END;
    END;
END;
PROCEDURE SELAL;
BEGIN
    WRITELN('1 MAPPING DIRECTO');
    WRITELN('2 MAX. DISTANCIA');
    WRITE('SELECCION = ');
    READLN(ALGO);
END;
PROCEDURE ENTRARVAL;
BEGIN
    WRITELN(' ');
    WRITELN('INDIQUE EL NUMERO DE VECES');
    WRITE('QUE DEBE EJECUTARSE UN BUCLE');
    WRITELN(', SUS LIMITES Y EL NUMERO');
    WRITELN('DE ACTIVACIONES');
    WRITELN(' ');
    REPEAT
        WRITE('DESDE DONDE ? ');
        READLN(ORG);
        WRITE('HASTA DONDE ? ');
        READLN (DEST);
        LISTA[ORG].SALTO:=DEST;
        WRITE ('CUANTAS VUELTAS ? ');
        READLN(VUEL);
        LISTA[ORG].VUELTAS:=VUEL;
        LISTA[ORG].CUVU := VUEL;
    UNTIL

```

```

    READLN(ALTO);
    UNTIL ALTO='Q';
END;
PROCEDURE BUSQUEDA;
VAR
    A,B: INTEGER;
BEGIN
    ESTA := FALSE;
    IF POPRO MOD TAMBO = 0 THEN POBO := TAMBO
    ELSE POBO := POPRO MOD TAMBO;
    FOR A := 1 TO NUMBO DO
        BEGIN
            B := POBO + TAMBO * (A-1);
            IF MEMIC[B] = POPRO THEN ESTA:= TRUE;
        END;
    END;
PROCEDURE MAPDIREC;
BEGIN
    IF PAG MOD NUMBO = 0 THEN BLOQUE := NUMBO
    ELSE BLOQUE := PAG MOD NUMBO;
END;
PROCEDURE HISTORIA;
VAR
    B,R: INTEGER;
BEGIN
    EXBO := FALSE;
    FOR R := 1 TO NUMREG DO
        BEGIN
            FOR B := 1 TO NUMBO DO
                BEGIN
                    PAG := (PROPO-1) DIV TAMBO + 1;
                    IF REGHISTORIA[B,R] = PAG THEN
                        BEGIN
                            EXBO := TRUE;
                            BLOQUE := B;
                        END;
                END;
            END;
        END;
    END;
PROCEDURE LEJOS;
VAR
    A,B,C,PRIMER,SEGON : INTEGER;
BEGIN
    C := 0;
    PRIMER := 1;
    SEGON := 0;
    FOR B := 1 TO NUMBO DO

```

```

BEGIN
  A := ABS(PAG - REGHISTORIA[B,1]);
  IF A > C THEN
    BEGIN
      C := A;
      PRIMER := B;
      SEGON := 0;
    END
  ELSE IF A=C THEN SEGON := B;
  END;
  IF SEGON = 0 THEN BLOQUE := PRIMER
  ELSE IF REGHISTORIA[PRIMER,1] < REGHISTORIA[SEGON,1] THEN BLOQUE :=
    ELSE BLOQUE := SEGON;
  END;
PROCEDURE ACTUALHIS;
VAR
  R: INTEGER;
BEGIN
  FOR R := NUMREG DOWNT0 1 DO
    BEGIN
      IF R=1 THEN REGHISTORIA[BLOQUE,R]:= PAG
      ELSE REGHISTORIA[BLOQUE,R] := REGHISTORIA[BLOQUE,R-1];
    END;
  END;
PROCEDURE LAZOS;
VAR
  A : INTEGER;
BEGIN
  WRITELN(CHR(12));
  CASE ALGO OF
    1 : WRITELN('MAPPING DIRECTO');
    2 : WRITELN('ALGORITMO DISTANCIA');
  END;
  WRITE('HISTOR = ', NUMREG);
  WRITELN(' TAM. TOTAL M.I. = ', TAMTOMI);
  WRITE('PROGRAMA DE ', LONGLIS);
  WRITELN(' CON LOS LAZOS :');
  FOR A:=1 TO LONGLIS DO
    BEGIN
      IF LISTA[A].SALTO <> 0 THEN
        BEGIN
          WRITELN(LISTA[A].SALTO:4, ' <--- ', A:4,
            ' (' , LISTA[A].VUELTAS:4, ')');
        END;
      END;
    REPEAT
  UNTIL KEYPRESS;

```

```

END;
PROCEDURE IMPRIMIR;
VAR
Q : FILE OF CHAR;
A : INTEGER;
BEGIN
REWRITE(Q, 'PRINTER:PP');
WRITE(Q, 'DETALLMI ');
CASE ALGO OF
1 : WRITE(Q, 'MAPPING DIRECTO');
2 : WRITE(Q, 'ALGORITMO DISTANCIA');
END;
WRITELN(Q, ' TAMA/O TOTAL M. I. = ', TAMTOMI);
WRITE(Q, 'NORMAL = ', NUMINST);
WRITELN(Q, ' HISTOR = ', NUMREG);
WRITE(Q, 'PROGRAMA DE ', LONGLIS);
WRITELN(Q, ' CON LOS LAZOS :');
FOR A:=1 TO LONGLIS DO
BEGIN
IF LISTA[A].SALTO <> 0 THEN
BEGIN
WRITELN(Q, LISTA[A].SALTO:4, ' <--- ', A:4,
' (', LISTA[A].VUELTAS:4, ')');
END;
END;
FOR NUMBO := 1 TO TAMTOMI DO
BEGIN
IF TAMTOMI MOD NUMBO = 0 THEN
BEGIN
TAMBO := TAMTOMI DIV NUMBO;
WRITELN(Q, 'CON ', NUMBO, ' BLOQUES DE ', TAMBO);
WRITELN(Q, 'EN M.I. = ', NUMINST/NUMBO, ' GANANCIA = ',
GANAN[NUMBO], ' %');
WRITELN(Q, ' ');
END;
END;
WRITELN(Q, ' ');
WRITELN(Q, ' ');
CLOSE(Q, LOCK);
END;
PROCEDURE RESULTADOS;
VAR
A : INTEGER;
BEGIN
WRITELN(CHR(12));
WRITE('DETALLMI : ');
CASE ALGO OF

```

```

1 : WRITELN('MAPPING DIRECTO');
2 : WRITELN('ALGORITMO DISTANCIA');
END;
WRITE('NORMAL = ', NUMINST, ' ');
WRITELN(' HISTOR = ', NUMREG, ' ');
WRITE('PROGRAMA DE ', LONGLIS);
WRITELN(' CON LOS LAZOS :');
WRITELN(' ');
FOR A:=1 TO LONGLIS DO
  BEGIN
    IF LISTA[A].SALTO <> 0 THEN
      BEGIN
        WRITELN(LISTA[A].SALTO:4, ' <--- ', A:4,
          ' (', LISTA[A].VUELTAS:4, ')');
      END;
    END;
  END;
FOR NUMBO := 1 TO TAMTOMI DO
  BEGIN
    IF TAMTOMI MOD NUMBO = 0 THEN
      BEGIN
        TAMBO := TAMTOMI DIV NUMBO;
        WRITELN('CON ', NUMBO, ' BLOQUES DE ', TAMBO);
        WRITELN('EN M.I. = ', NUMMI[ NUMBO ], ' GANANCIA = ',
          GANAN[ NUMBO ], ' %');
        WRITELN(' ');
      END;
    END;
  END;
WRITELN(' ');
WRITELN(' ');
WRITE('IMPRIMIR = P ');
READLN(ALTO);
IF ALTO = 'P' THEN IMPRIMIR;
END;
PROCEDURE MENU;
BEGIN
  WHILE SEL < 6 DO
    BEGIN
      WRITELN(CHR(12));
      WRITELN(' ');
      WRITELN('0  VISUALIZACION ESTADO');
      WRITELN(' ');
      WRITELN('1  CAMBIAR BLOQUES M.I. ');
      WRITELN(' ');
      WRITELN('2  NUM. REGIS. HIST. ');
      WRITELN(' ');
      WRITELN('3  SALTOS.           ');
      WRITELN(' ');
      WRITELN('4  RESET.           ');
      WRITELN(' ');
      WRITELN('5  SELECCION ALGORITMO. ');
    END;
  END;

```

```

WRITELN(' ');
WRITELN('6' EJECUTAR ');
WRITELN(' ');
WRITELN('7' TERMINAR. ');
WRITELN(' ');
WRITE(' SELECCION: ');
READLN(SEL);
CASE SEL OF
  0 : LAZOS;
  1 : BEGIN
      MEMINTER;
      INIMEMI;
      END;
  2 : BEGIN
      REGISTROS;
      END;
  3 : ENTRARVAL;
  4 : BEGIN
      LONGLISTA;
      MEMINTER;
      REGISTROS;
      INICIALISTA;
      INIMEMI;
      ENTRARVAL;
      SELAL;
      END;
  5 : SELAL;
  6,7:;
END;
END;
END;

BEGIN
  SEL := 4;
  MENU;
  WHILE SEL < 7 DO
  BEGIN
    FOR NUMBO := 1 TO TAMTOMI DO
    BEGIN
      IF TAMTOMI MOD NUMBO = 0 THEN
      BEGIN
        TAMBO := TAMTOMI DIV NUMBO;
        INIMEMI;
        INIREG;
        POPRO:=1;
        NUMINST:=0;
        NUINMI[ NUMBO ]:=0;
      END;
    END;
  END;
END;

```

```

WHILE POPRO <= LONGLIS DO
  BEGIN
    NUMINST := NUMINST +1;
    BUSQUEDA;
    IF ESTA = TRUE THEN NUINMI[NUMBO] := NUINMI[NUMBO] + 1
    ELSE
      BEGIN
        HISTORIA;
        IF EXBO = FALSE THEN
          BEGIN
            CASE ALGO OF
              1 : MAPDIREC;
              2 : LEJOS;
            END;
          END;
          (* ESCRIBIR *);
          POSESCRI := POBO + TAMBO * (BLOQUE -1);
          MEMI[POSESCRI] := POPRO;
          (*SOLO HARIA FALTA ESCRIBIR LA PARTE ALTA DE POPRO*);
          ACTUALHIS;
        END;
        IF (LISTA[POPRO].SALTO=0) OR (LISTA[POPRO].CUVU=0) THEN
          BEGIN
            LISTA[POPRO].CUVU := LISTA[POPRO].VUELTAS;
            POPRO:=POPRO + 1;
          END
        ELSE
          BEGIN
            LISTA[POPRO].CUVU := LISTA[POPRO].CUVU - 1;
            POPRO := LISTA[POPRO].SALTO;
          END;
        END;
      END;
    GANAN[NUMBO] := (NUINMI[NUMBO]/NUMINST)*100;
  END;
  END;
  RESULTADOS;
  SEL := 4;
  MENU;
END;
END.

```


APENDICE C
LISTADO SIMULADOR PROBABILISTICO

BRANCIRU

```
(*$G+*)
(*$S+*)
PROGRAM BRANCLRU;
USES
  APFLESTUF;
LABEL 200;
TYPE
  ALGORITMO = 1..3;
  SELECCION = 0..8;
VAR
  ALGO : ALGORITMO;
  SEL : SELECCION;
  LISTA : ARRAY [1..1000] OF RECORD
  SALTO:INTEGER;
  PRO : REAL;
  END;
  ESTA,EXBO : BOOLEAN;
  REGHISTORIA: ARRAY [1..99,1..8] OF INTEGER;
  RELRU: ARRAY [1..99] OF INTEGER;
  MEMI : ARRAY [1..1000] OF INTEGER;
  BLOQUE, BRANC, TAMBUMAX, NUMINST, NUINMI, TAMBO,
  NUMBO, ORG, DEST, VUEL, LONGLIS, POSESCRI,
  PAG, NUMREG, POBO, POPRO, TAMTOMI,
  PAR, VECES, X, S, V : INTEGER;
  CH ,ALTO : CHAR;
  NUSU, NUTO, NUMISU, NUMITO, PROB,
  P, ALEA, GANAN : REAL;
PROCEDURE LONGLISTA; (*PIDE LA LONGITUD DEL PROGRAMA*)
BEGIN
  WRITE (' ] LONGITUD DEL PROGRAMA ? = ');
  READLN (LONGLIS);
  WRITELN(' ');
END;
PROCEDURE MEMINTER;
BEGIN
  WRITE (' ] BLOQUES DE MEM. DE INTERC. ? ');
  READLN(NUMBO);
  WRITE (' ] TAMA/O EN BYTES DE CADA BLOQUE ? ');
  READLN(TAMBO);
END;
PROCEDURE REGISTROS;
BEGIN
  WRITE(' ] CUANTOS REGISTROS DE HISTORIA ? ');
  READLN(NUMREG);
END;
PROCEDURE INICIALISTA; (*PONE TODO EL PROGRAMA INICIALMENTE A CERO*)
```

```

VAR A : INTEGER;
BEGIN
  FOR A := 1 TO LONGLIS DO
    BEGIN
      WITH LISTA[A] DO
        BEGIN
          SALTO := 0;
          PRO := 0;
        END;
      END;
    END;
  END;
PROCEDURE INIMEMI;
VAR A : INTEGER;
BEGIN
  TAMTOMI:=TAMBO*NUMBO;
  FOR A:=1 TO TAMTOMI DO
    BEGIN
      MEMI[A]:=0;
    END;
  END;
PROCEDURE INIREG;
VAR R,B : INTEGER;
BEGIN
  FOR R:=1 TO NUMREG DO
    BEGIN
      FOR B:=1 TO NUMBO DO
        BEGIN
          REGHISTORIA[B,R]:=-B;
        END;
      END;
    END;
  END;
PROCEDURE INILRU;
VAR
  A : INTEGER;
BEGIN
  FOR A := 1 TO NUMBO DO
    BEGIN
      RELRU[A] := A;
    END;
  END;
PROCEDURE ACTUALRU;
VAR
  A , SALVA,LUGAR : INTEGER;
BEGIN
  FOR A:=1 TO NUMBO DO
    BEGIN
      IF RELRU[A]=BLOQUE THEN
        BEGIN

```

```

        LUGAR:=A;
        SALVA := RELRUC[A];
    END;
END;
FOR A := LUGAR DOWNTO 2 DO
    BEGIN
        RELRUC[A] := RELRUC[A-1];
    END;
    RELRUC[1] := SALVA;
END;
PROCEDURE SELAL;
    BEGIN
        WRITELN('1 MAPPING DIRECTO. ');
        WRITELN('2 DISTANCIA. ');
        WRITELN('3 L. R. U. ');
        WRITE('SELECCION = ');
        READLN(ALGO);
        IF ALGO = 3 THEN INILRU;
    END;
PROCEDURE ENTRARVAL;
    BEGIN
        WRITELN(' ');
        WRITE(' V , P ');
        READLN(ALTO);
        WRITELN(' ');
        REPEAT
            WRITE('DESDE DONDE ? ');
            READLN(ORG);
            WRITE('HASTA DONDE ? ');
            READLN (DEST);
            LISTA[ORG].SALTO:=DEST;
            IF ALTO = 'V' THEN
                BEGIN
                    WRITE ('CUANTAS VUELTAS ? ');
                    READLN(VUEL);
                    PROB := VUEL/(VUEL+1);
                END
            ELSE
                BEGIN
                    WRITE('PROBABILIDAD ? ');
                    READLN(PROB);
                END;
            LISTA[ORG].PRO := PROB;
            WRITE(' V , P , S ');
            READLN(ALTO);
        UNTIL ALTO='S';
    END;
PROCEDURE ENTRAVECES;

```

```

BEGIN
  WRITELN(' CUANTAS VECES QUIERE EJECUTAR');
  WRITE(' EL PROGRAMA ? ');
  READLN(VECES);
END;
PROCEDURE GENNUAL;
VAR
  KAOS : INTEGER;
BEGIN
  (*RANDOMIZE*);
  KAOS := RANDOM;
  ALEA := KAOS / 32767;
END;
PROCEDURE BUSQUEDA;
VAR
  A,B: INTEGER;
BEGIN
  ESTA := FALSE;
  IF POPRO MOD TAMBO = 0 THEN POBO := TAMBO
  ELSE POBO := POPRO MOD TAMBO;
  FOR A := 1 TO NUMBO DO
    BEGIN
      B := POBO + TAMBO * (A-1);
      IF MEMI[B] = POPRO THEN ESTA:= TRUE;
      BLOQUE := A;
    END;
  END;
END;
PROCEDURE MAPDIREC;
BEGIN
  IF PAG MOD NUMBO = 0 THEN BLOQUE := NUMBO
  ELSE BLOQUE := PAG MOD NUMBO;
END;
PROCEDURE LRU;
BEGIN
  BLOQUE := RELRU[NUMBO];
END;
PROCEDURE HISTORIA;
VAR
  B,R: INTEGER;
BEGIN
  EXBO := FALSE;
  FOR R := 1 TO NUMREG DO
    BEGIN
      FOR B := 1 TO NUMBO DO
        BEGIN
          PAG := (POPRO-1) DIV TAMBO + 1;
          IF REGHISTORIA[B,R] = PAG THEN
            BEGIN
              EXBO := TRUE;
            END;
        END;
      END;
    END;
  END;

```

```

        BLOQUE := B;
      END;
    END;
  END;
END;
PROCEDURE DISTANCIA;
VAR
  A,B,C,PRIMER,SEGON : INTEGER;
BEGIN
  C := 0;
  PRIMER:=1;
  SEGON:=0;
  FOR B := 1 TO NUMBO DO
    BEGIN
      A := ABS(PAG - REGHISTORIA[B,1]);
      IF A > C THEN
        BEGIN
          C := A;
          PRIMER:=B;
          SEGON:=0;
        END
      ELSE IF A=C THEN SEGON :=B;
    END;
  IF SEGON = 0 THEN BLOQUE := PRIMER
  ELSE IF REGHISTORIA[PRIMER,1]<REGHISTORIA[SEGON,1] THEN BLOQUE := PR.
  ELSE BLOQUE := SEGON;
END;
PROCEDURE ACTUALHIS;
VAR
  R: INTEGER;
BEGIN
  FOR R := NUMREG DOWNT0 1 DO
    BEGIN
      IF R=1 THEN REGHISTORIA[BLOQUE,R]:= PAG
      ELSE REGHISTORIA[BLOQUE,R] := REGHISTORIA[BLOQUE,R-1];
    END;
  END;
END;
PROCEDURE LAZOS;
VAR
  A : INTEGER;
BEGIN
  WRITELN(CHR(12));
  CASE ALGO OF
    1 : WRITELN('MAPPING DIRECTO');
    2 : WRITELN('ALGORITMO DISTANCIA');
    3 : WRITELN('L. R. U. ');
  END;
  WRITELN('CON ',NUMREG,' REGISTROS DE HISTORIA');

```

```

WRITELN(' ');
WRITELN('CON ', NUMBO, ' BLOQUES DE ', TAMBO, ' BYTES DE M. I. ');
WRITELN(' ');
WRITELN('EN UN PROGRAMA DE ', LONGLIS, ' INSTRUCCIONES ');
WRITELN(' ');
WRITELN('CON LOS LAZOS : ');
WRITELN(' ');
FOR A:=1 TO LONGLIS DO
  BEGIN
    IF LISTA[A].SALTO <> 0 THEN
      BEGIN
        S := LISTA[A].SALTO;
        P := LISTA[A].PRO;
        IF P = 1 THEN V := 32767
          ELSE V := ROUND(P/(1-P));
        WRITELN(S:4, ' <--- ', A:4,
          ' (', P, ')', ' (', V, ')');
      END;
    END;
  WRITELN('QUE SE EJECUTARA ', VECES, ' VECES ');
  GOTOXY(37, 13);
  WRITELN(NUMREG, 'R.H. ');
  GOTOXY(37, 14);
  WRITELN(NUMBO, 'DE', TAMBO);
  GOTOXY(37, 15);
  WRITELN(LONGLIS, ' INST ');
  GOTOXY(37, 16);
  WRITELN(VECES, ' VECES ');
  GOTOXY(37, 17);
  CASE ALGO OF
    1 : WRITELN('DIREC ');
    2 : WRITELN('DISTA ');
    3 : WRITELN('LRU ');
  END;
END;
PROCEDURE IMPRIMIR;
VAR
  Q : FILE OF CHAR;
  A : INTEGER;
BEGIN
  REWRITE(Q, 'PRINTER:PP ');
  WRITE(Q, 'BRANCMI ');
  CASE ALGO OF
    1 : WRITELN(Q, 'MAPPING DIRECTO ');
    2 : WRITELN(Q, 'ALGORITMO DISTANCIA ');
    3 : WRITELN(Q, 'L. R. U. ');
  END;
  WRITELN(Q, 'EJECUCION NORMAL = ', NUTO,
    ' INSTRUCCIONES ');
  WRITELN(Q, 'EN M. I. = ', NUMITO, ' INSTRUCCIONES ');

```

```

WRITELN(Q,'GANANCIA = ',GANAN,' %');
WRITELN(Q,'CON ',NUMREG,' REGISTROS DE HISTORIA');
WRITELN(Q,'CON ',NUMBO,' BLOQUES DE ',TAMBO,' BYTES DE M. I. ');
WRITELN(Q,'EN UN PROGRAMA DE ',LONGLIS,' INSTRUCCIONES');
WRITELN(Q,'QUE SE HA EJECUTADO ',VECES,' VECES');
WRITELN(Q,'CON LOS LAZOS :');
WRITELN(Q,' ');
FOR A:=1 TO LONGLIS DO
  BEGIN
    IF LISTA[A].SALTO <> 0 THEN
      BEGIN
        S := LISTA[A].SALTO;
        P := LISTA[A].PRO;
        IF P = 1 THEN V := 32767
          ELSE V := ROUND(P/(1-P));
        WRITELN(Q,S:4,' <--- ',A:4,
          ' (',P,')', ' (',V,')');
      END;
    END;
    WRITELN(Q,' ');
    WRITELN(Q,' ');
    WRITELN(Q,' ');
    CLOSE(Q,LOCK);
  END;
PROCEDURE RESULTADOS;
VAR
  A : INTEGER;
BEGIN
  WRITELN(CHR(12));
  CASE ALGO OF
    1 : WRITELN('MAPPING DIRECTO');
    2 : WRITELN('ALGORITMO DISTANCIA');
    3 : WRITELN('L. R. U. ');
  END;
  WRITELN('EJECUCION NORMAL = ',NUTO,
  ' INSTRUCCIONES');
  WRITELN('EN M. I. = ',NUMITO,' INSTRUCCIONES');
  WRITELN('GANANCIA = ',GANAN,' %');
  WRITELN('CON ',NUMREG,' REGISTROS DE HISTORIA');
  WRITELN('CON ',NUMBO,' BLOQUES DE ',TAMBO,' BYTES DE M. I. ');
  WRITELN('EN UN PROGRAMA DE ',LONGLIS,' INSTRUCCIONES');
  WRITELN('QUE SE HA EJECUTADO ',VECES,' VECES');
  WRITELN('CON LOS LAZOS :');
  WRITELN(' ');
  FOR A:=1 TO LONGLIS DO
    BEGIN
      IF LISTA[A].SALTO <> 0 THEN
        BEGIN
          S := LISTA[A].SALTO;
          P := LISTA[A].PRO;

```

```

        IF P = 1 THEN V := 32767
        ELSE V := ROUND(P/(1-P));
        WRITELN(S:4,' <--- ',A:4,
        ' (',P,')',',', ' (',V,')');
    END;
END;
WRITELN(' ');
WRITELN(' ');
NOTE (30,50);
NOTE (35,50);
NOTE (30,50);
NOTE (35,50);
WRITE('IMPRIMIR = P ');
READLN(ALTO);
IF ALTO = 'P' THEN IMPRIMIR;
END;
PROCEDURE PARCIAL;
VAR
    Q : FILE OF CHAR;
BEGIN
    NUTO := NUSU / PAR;
    NUMITO := NUMISU / PAR;
    GANAN := (NUMITO/NUTO)*100;
    REWRITE(Q,'PRINTER:PP');
    WRITE(Q,PAR,' BR ');
    CASE ALGO OF
        1 : WRITE(Q,'DIR ');
        2 : WRITE(Q,'DIS ');
        3 : WRITE(Q,'LRU. ');
    END;
    WRITE(Q,NUMBO,'DE',TAMBO);
    WRITE(Q,' N = ',NUTO,' ');
    WRITE(Q,'MI = ',NUMITO,' ');
    WRITELN(Q,'G = ',GANAN,'%');
    CLOSE(Q,LOCK);
END;
PROCEDURE MENU;
BEGIN
    WHILE SEL < 7 DO
        BEGIN
            WRITELN(CHR(12));
            WRITELN(' ');
            WRITELN('0  VISUALIZACION ESTADO');
            WRITELN(' ');
            WRITELN('1  CAMBIAR BLOQUES M.I. ');
            WRITELN(' ');
            WRITELN('2  NUM. REGIS. HIST. ');
            WRITELN(' ');
            WRITELN('3  SALTOS. ');
            WRITELN(' ');

```



```

WRITELN('4  VECES EJECUCION. ');
WRITELN(' ');
WRITELN('5  RESET.           ');
WRITELN(' ');
WRITELN('6  SELECCION ALGORITMO. ');
WRITELN(' ');
WRITELN('7  EJECUTAR           ');
WRITELN(' ');
WRITELN('8  TERMINAR.           ');
WRITELN(' ');
WRITE('                               SELECCION: ');
READLN(SEL);
CASE SEL OF
  0 : BEGIN
      LAZOS;
      REPEAT
      UNTIL KEYPRESS;
      END;
  1 : BEGIN
      MEMINTER;
      INIMEMI;
      END;
  2 : BEGIN
      REGISTROS;
      INIREG;
      END;
  3 : ENTRARVAL;
  4 : ENTRAVECES;
  5 : BEGIN
      LONGLISTA;
      MEMINTER;
      REGISTROS;
      INICIALISTA;
      INIMEMI;
      INIREG;
      ENTRARVAL;
      SELAL;
      ENTRAVECES;
      END;
  6 : SELAL;
  7 : BEGIN
      LAZOS;
      GOTOXY(0,23);
      WRITE(VECES);
      END;
  8 : ;

END;
END;

```

```

END;
PROCEDURE EJECUCION;
BEGIN
  POPRO:=1;
  NUMINST:=0;
  NUINMI:=0;
  IF ALGO = 3 THEN INILRU;

  WHILE POPRO <= LONGLIS DO
  BEGIN
    NUMINST := NUMINST +1;
    BUSQUEDA;
    IF ESTA = TRUE THEN NUINMI := NUINMI + 1
    ELSE
      BEGIN
        HISTORIA;
        IF EXBO = FALSE THEN
          BEGIN
            CASE ALGO OF
              1 : MAPDIREC;
              2 : DISTANCIA;
              3 : LRU;
            END;
          END;
        (* ESCRIBIR *);
        POSESCRI := POBO + TAMBO * (BLOQUE -1);
        MEMI[POSESCRI]:= POPRO;
        (*SOLO HARIA FALTA ESCRIBIR LA PARTE ALTA DE POPRO*);
        ACTUALHIS;
      END;
    IF ALGO = 3 THEN ACTUALRU;
    IF LISTA[POPRO].SALTO=0 THEN POPRO := POPRO +1
    ELSE
      BEGIN
        (*RANDOMIZE*);
        GENNUAL;
        IF ALEA <= LISTA[POPRO].PRO THEN
          POPRO := LISTA[POPRO].SALTO
        ELSE POPRO := POPRO + 1;
      END;
    END; (* DEL WHILE *);
  END;

  BEGIN
    SEL := 6;
    MENU;
  
```

```

WHILE SEL < 8 DO
  BEGIN

    RANDOMIZE;
    NUSU :=0;
    NUMISU := 0;
    FOR X := 1 TO VECES DO
      BEGIN
        INIMEMI;
        INIREG;
        EJECUCION;
        NUSU := NUSU + NUMINST;
        NUMISU := NUMISU + NUINMI;
        GOTOXY(8,23);
        WRITE(X);
        IF KEYPRESS THEN
          BEGIN
            READ(KEYBOARD,CH);
            IF CH = 'S' THEN
              GOTO 200;
            GOTOXY(8,23);
            PAR:=X;
            PARCIAL;
          END;
        END;
      NUTO := NUSU / VECES;
      NUMITO := NUMISU / VECES;
      GANAN := (NUMITO/NUTO)*100;
      RESULTADOS;
      200 : SEL := 5;
      MENU;

    END;

  END.

```

APENDICE D

LISTADO DE MICROPROGRAMAS PARA EL MICROPROCESADOR R6502

A-1 1 byte - 2 ciclos Sin referencia a memoria

```

. . .
PEPE      TO - 1
          TO - 2
          TO - 3      Búsqueda
          TO - 4

          T1 - 1
1MAP      T1 - 2 - JMP 1MAP
          T1 - 3 - NOP
          T1 - 4 - JMP 2MAP -- Salto a la parte operativa
                                   (2a pregunta)

2MAP      TO - 1 - JMP PEPE // <PC> --> H,L // Parte ope-
                                   rativa
    
```

A-2-1 2 byte - 2 ciclos (inmediato)

```

PEPE      TO - 1
          TO - 2
          TO - 3      Búsqueda
          TO - 4

          T1 - 1 <PC> --> H,L
          T1 - 2 JMP 1MAP

1MAP      T1 - 3 INC PC
          T1 - 4 <M> --> L1 // JMP 2MAP

2MAP      TO - 1 JMP PEPE // <PC> --> H,L // Parte opera-
                                   tiva
    
```

A-2-2 2 byte - 3 ciclos

```

PEPE      TO - 1
          TO - 2
          TO - 3      Búsqueda
          TO - 4

          T1 - 1 <PC> --> H,L
    
```

```

T1 - 2  JMP 1MAP
1MAP    T1 - 3  INC PC
        T1 - 4  <M> --> L1 (ADL) // #0 --> H

        T2 - 1  <L1> --> L
        T2 - 2  NOP
        T2 - 3  NOP
        T2 - 4  <M> --> L1 (Dato) // JMP 2MAP
2MAP    T0 - 1  JMP PEPE // <PC> --> H,L // Parte operativa

```

A-2-3 3 byte - 4 ciclos (Absoluto)

```

PEPE    T0 - 1
        T0 - 2
        T0 - 3  Búsqueda
        T0 - 4

        T1 - 1  <PC> --> H,L
        T1 - 2  JMP 1MAP
1MAP    T1 - 3  INC PC
        T1 - 4  <M> --> L1 (ADL)

        T2 - 1  <PC> --> HL
        T2 - 2  NOP
        T2 - 3  INC PC
        T2 - 4  <M> --> L1 (ADH) // <L1> --> L

        T3 - 1  <L1> --> H
        T3 - 2  NOP
        T3 - 3  NOP
        T3 - 4  <M> --> L1 (Dato) // JMP 2MAP
2MAP    T0 - 1  <PC> --> H,L // JMP PEPE // Parte operativa

```

A-2-4 2 byte - 6 ciclos (Indirecto, X Pag. Cero)

```

PEPE    T0 - 1
        T0 - 2
        T0 - 3  Búsqueda
        T0 - 4

        T1 - 1  <PC> --> H,L
        T1 - 2  JMP 1MAP
1MAP    T1 - 3  INC PC
        T1 - 4  <M> --> L1 (BAL) // # 0 --> H

```

```

T2 - 1 <L1> --> L
T2 - 2 NOP
T2 - 3 NOP
T2 - 4 0 -> C(UAL) (Función UAL).

T3 - 1 <L1> + X --> L,L1
T3 - 2 NOP
T3 - 3 0 -> C(UAL)
T3 - 4 <M> --> L1 (ADL) // <L1> + 1 --> L

T4 - 1 NOP
T4 - 2 NOP
T4 - 3 NOP
T4 - 4 <M> --> L1 (ADH) // <L1> --> L (ADL)

T5 - 1 <L1> --> H
T5 - 2 NOP
T5 - 3 NOP
T5 - 4 <M> --> L1 (Dato) // JMP 2MAP

```

```

2MAP    T0 - 1 <PC> --> H,L // JMP PEPE // Parte operativa

```

A-2-5 3 byte - 4 ó 5 ciclos (Absoluto, X ó Y)

```

PEPE    T0 - 1
        T0 - 2
        T0 - 3   Búsqueda
        T0 - 4

        T1 - 1 <PC> --> H,L
        T1 - 2   JMP 1MAP

1MAP    T1 - 3   INC PC
        T1 - 4 <M> --> L1 (BAL)

        T2 - 1 <PC> --> H,L
        T2 - 2   INC PC
        T2 - 3   0 -> C(UAL)
        T2 - 4 <M> --> L1 (BAH) // <L1> + X ó Y --> L

        T3 - 1 <L1> --> H // IF C(UAL) = 0 THEN GO TO
                        ZORRO
        T3 - 2   NOP
        T3 - 3   NOP
        T3 - 4   NOP

ZORRO   T4 - 1 <L1> + C(UAL) --> H
        T4 - 2   NOP
        T4 - 3   NOP
        T4 - 4 <M> --> L1 (Dato) // JMP 2MAP

```



```

T4 - 1 <L1> --> H // IF C(UAL) = 0 THEN GO TO
                MARTA
T4 - 2 NOP
T4 - 3 NOP
T4 - 4 <L1> + C(UAL) --> H

MARTA
T5 - 1 NOP
T5 - 2 NOP
T5 - 3 NOP
T5 - 4 <M> --> L1 (Dato) // JMP 2MAP

2MAP
T0 - 1 <PC> --> H,L // JMP PEPE // Parte operativa

```

A-3-1 STA/X/Y 2 byte - 3 ciclos (Pag. Cero)

```

PEPE
T0 - 1
T0 - 2
T0 - 3 Búsqueda
T0 - 4

T1 - 1 <PC> --> H,L
T1 - 2 JMP 1MAP

1MAP
T1 - 3 INC PC
T1 - 4 <M> --> L1 (ADL) // # 0 --> H

T2 - 1 <L1> --> L // WRITE
T2 - 2 <A/X/Y> --> L1 // WRITE
T2 - 3 NOP // WRITE
T2 - 4 NOP // JMP 2MAP WRITE

2MAP
T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa

```

A-3-2 3 byte - 4 ciclos (Absoluto) STORE

```

PEPE
T0 - 1
T0 - 2
T0 - 3 Búsqueda
T0 - 4

T1 - 1 <PC> --> H,L
T1 - 2 JMP 1MAP

1MAP
T1 - 3 INC PC
T1 - 4 <M> --> L1 (ADL)

T2 - 1 <PC> --> H,L
T2 - 2 NOP
T2 - 3 INC PC
T2 - 4 <M> --> L1 (ADH) // <L1> --> L

```


T3 - 1	<L1> --> H	WRITE
T3 - 2	<A/X/Y> --> L1	WRITE
T3 - 3	NOP	WRITE
T3 - 4	JMP 2MAP	WRITE

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa

A-3-3 2 byte - 6 ciclos (Indexado X, Indirecto) STORE

PEPE	T0 - 1		
	T0 - 2		
	T0 - 3	Búsqueda	
	T0 - 4		
1MAP	T1 - 1	<PC> --> H,L	
	T1 - 2	JMP 1MAP	
	T1 - 3	INC PC	
	T1 - 4	<M> --> L1 (BAL) // # 0 --> H	
	T2 - 1	<L1> --> L	
	T2 - 2	NOP	
	T2 - 3	NOP	
	T2 - 4	0 --> C(UAL)	
	T3 - 1	<L1> + X --> L,L1	
	T3 - 2	NOP	
	T3 - 3	0 --> C(UAL)	
	T3 - 4	<M> --> L1 (ADL) // <L1> + 1 --> L	
	T4 - 1	NOP	
	T4 - 2	NOP	
	T4 - 3	NOP	
	T4 - 4	<M> --> L1 (ADH) // <L1> --> L (ADL)	
	T5 - 1	<L1> --> H //	WRITE
	T5 - 2	<A> --> L1	WRITE
	T5 - 3	NOP	WRITE
	T5 - 4	JMP 2MAP	WRITE
2MAP	T0 - 1	<PC> --> H,L // JMP PEPE // Sin Parte operativa	

A-3-4 3 byte - 5 ciclos (Absoluto, X @ Y) STORE

PEPE	T0 - 1	
	T0 - 2	
	T0 - 3	Búsqueda
	T0 - 4	

```

T1 - 1 <PC> --> H,L
T1 - 2 JMP 1MAP
1MAP T1 - 3 INC PC

T1 - 4 <M> --> L1 (BAL)

T2 - 1 <PC> --> H,L
T2 - 2 INC PC
T2 - 3 0 --> C(UAL)
T2 - 4 <M> --> L1 (BAH) // <L1> --> L

T3 - 1 <L1> --> H
T3 - 2 NOP
T3 - 3 NOP
T3 - 4 <L> + X & Y + C(UAL) --> L

T4 - 1 <L1> + C(UAL) --> H WRITE
T4 - 2 <A> --> L1 WRITE
T4 - 3 NOP WRITE
T4 - 4 JMP 2MAP WRITE

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte ope-
rativa

```

A-3-5 2 byte - 4 ciclos (Indexado pag. cero X & Y) STORE

```

PEPE T0 - 1
T0 - 2
T0 - 3 Búsqueda
T0 - 4

T1 - 1 <PC> --> H,L
T1 - 2 JMP 1MAP

1MAP T1 - 3 INC PC
T1 - 4 <M> --> L1 (BAL) // # 0 --> H

T2 - 1 <L1> --> L
T2 - 2 NOP
T2 - 3 NOP
T2 - 4 # 0 --> C(UAL)

T3 - 1 <L1> + X, Y --> L WRITE
T3 - 2 <A> --> L1 WRITE
T3 - 3 NOP WRITE
T3 - 4 JMP 2MAP WRITE

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte ope-
rativa.

```

A-3-6 2 byte - 6 ciclos (Indirecto indexado Y) STORE

```

PEPE      T0 - 1
          T0 - 2
          T0 - 3      Búsqueda
          T0 - 4

          T1 - 1 <PC> --> H,L
          T1 - 2 JMP 1MAP

1MAP      T1 - 3 INC PC
          T1 - 4 <M> --> L1 (IAL) // # 0 --> H

          T2 - 1 <L1> --> L
          T2 - 2 NOP
          T2 - 3 # 0 --> C(UAL)
          T2 - 4 <M> --> L1 (BAL) // <L1> + # 1 --> L

          T3 - 1 NOP
          T3 - 2 NOP
          T3 - 3 #0 --> C(UAL)
          T3 - 4 <M> --> L1 (BAH) // <L1> + Y --> L

          T4 - 1 NOP
          T4 - 2 NOP
          T4 - 3 NOP
          T4 - 4 NOP

          T5 - 1 <L1> --> H                WRITE
          T5 - 2 <A> --> L1                WRITE
          T5 - 3 NOP                        WRITE
          T5 - 4 JMP 2MAP                  WRITE

2MAP      T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte ope-
                                               rativa.

```

A-4-1 2 byte - 5 ciclos (Página Cero) MODIFY

```

PEPE      T0 - 1
          T0 - 2
          T0 - 3      Búsqueda
          T0 - 4

          T1 - 1 <PC> --> H,L
          T1 - 2 JMP 1MAP

1MAP      T1 - 3 INC PC
          T1 - 4 <M> --> L1 (ADL) // # 0 --> H

          T2 - 1 <L1> --> L
          T2 - 2 NOP
          T2 - 3 NOP
          T2 - 4 <M> --> L1 (Dato)

```

```

T3 - 1  NOP
T3 - 2  NOP
T3 - 3  NOP
T3 - 4  JMP 2MAP

2MAP    T4 - 1  PARTE OPERATIVA                WRITE
        T4 - 2  DATO MODIFICADO --> L1        WRITE
        T4 - 3  NOP                            WRITE
        T4 - 4  JMP TO - 1                    WRITE

        TO - 1  <PC> --> H,L // JMP PEPE // Sin Parte oper-
                                                rativa.

```

A-4-2 3 byte - 6 ciclos (Absoluto) MODIFY

```

PEPE    TO - 1
        TO - 2
        TO - 3  Búsqueda
        TO - 4

        T1 - 1  <PC> --> H,L
        T1 - 2  JMP 1MAP

1MAP    T1 - 3  INC PC

        T1 - 4  <M> --> L1 (ADL)

        T2 - 1  <PC> --> H,L
        T2 - 2  NOP
        T2 - 3  INC PC
        T2 - 4  <M> --> L1 (ADH) // <L1> --> L

        T3 - 1  <L1> --> H
        T3 - 2  NOP
        T3 - 3  NOP
        T3 - 4  <M> --> L1 (Dato)

        T4 - 1  NOP
        T4 - 2  NOP
        T4 - 3  NOP
        T4 - 4  JMP 2MAP

2MAP    T5 - 1  PARTE OPERATIVA                WRITE
        T5 - 2  DATO MODIFICADO --> L1        WRITE
        T5 - 3  NOP                            WRITE
        T5 - 4  JMP TO - 1                    WRITE

        TO - 1  <PC> --> H,L // JMP PEPE // Sin Parte oper-
                                                rativa.

```

A-4-3 2 byte - 6 ciclos (Indexado Pag. Cero, X) MODIFY

```

PEPE      T0 - 1
          T0 - 2
          T0 - 3      Búsqueda
          T0 - 4

          T1 - 1      <PC> --> H,L
          T1 - 2      JMP 1MAP

1MAP      T1 - 3      INC PC
          T1 - 4      <M> --> L1 (BAL) // # 0 --> H

          T2 - 1      <L1> --> L
          T2 - 2      NOP
          T2 - 3      NOP
          T2 - 4      # 0 --> C(UAL)

          T3 - 1      <L1> + X --> L
          T3 - 2      NOP
          T3 - 3      NOP
          T3 - 4      <M> --> L1 (Dato)

          T4 - 1      NOP
          T4 - 2      NOP
          T4 - 3      NOP
          T4 - 4      JMP 2MAP

2MAP      T5 - 1      PARTE OPERATIVA                WRITE
          T5 - 2      DATO MODIFICADO --> L1          WRITE
          T5 - 3      NOP                            WRITE
          T5 - 4      JMP T0 - 1                      WRITE

          T0 - 1      <PC> --> H,L // JMP PEPE // Sin Parte ope-
                                                    rativa.

```

A-4-4 3 byte - 7 ciclos (Absoluto, X) MODIFY

```

PEPE      T0 - 1
          T0 - 2
          T0 - 3      Búsqueda
          T0 - 4

          T1 - 1      <PC> --> H,L
          T1 - 2      JMP 1MAP

1MAP      T1 - 3      INC PC
          T1 - 4      <M> --> L1 (BAL)

          T2 - 1      <PC> --> H,L
          T2 - 2      INC PC
          T2 - 3      # 0 --> C(UAL)
          T2 - 4      <M> --> L1 (BAH) // <L1> --> L

```

```

T3 - 1 <L1> --> H
T3 - 2 NOP
T3 - 3 NOP
T3 - 4 <L> + X --> L

T4 - 1 <H> + C(UAL) --> H
T4 - 2 NOP
T4 - 3 NOP
T4 - 4 <M> --> L1 (Dato)

T5 - 1 NOP
T5 - 2 NOP
T5 - 3 NOP
T5 - 4 JMP 2MAP

```

```

2MAP      T6 - 1 PARTE OPERATIVA           WRITE
          T6 - 2 DATO MODIFICADO --> L1    WRITE
          T6 - 3 NOP                       WRITE
          T6 - 4 JMP T0 - 1                WRITE

          T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte oper-
                                     rativa.

```

A-5-1 1 byte - 3 ciclos PUSH (PHP, PHA)

```

PEPE      T0 - 1
          T0 - 2
          T0 - 3 Búsqueda
          T0 - 4

          T1 - 1 <PC> --> H,L
          T1 - 2 JMP 1MAP

1MAP      T1 - 3 NOP
          T1 - 4 <M> --> L1 (Código descartado) // #1 --> H

          T2 - 1 <S> --> L // JMP 2MAP           WRITE

2MAP      T2 - 2 <A,P> --> L1                   WRITE
          T2 - 3 <S> - 1 --> S                 WRITE
          T2 - 4 JMP T0 - 1                   WRITE

          T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte oper-
                                     rativa.

```

A-5-2 1 byte - 4 ciclos PULL (PLA, PLP)

PEPE	T0 - 1	
	T0 - 2	
	T0 - 3	Búsqueda
	T0 - 4	
	T1 - 1	<PC> --> H, L
	T1 - 2	JMP 1MAP
1MAP	T1 - 3	NOP
	T1 - 4	# 1 --> H
	T2 - 1	NOP
	T2 - 2	NOP
	T2 - 3	NOP
	T2 - 4	# 0 --> C(UAL)
	T3 - 1	<S> + 1 --> S, L
	T3 - 2	NOP
	T3 - 3	NOP
	T3 - 4	<M> --> L1 (Dato) // JMP 2MAP
2MAP	T0 - 1	<PC> --> H, L // JMP PEPE // Sin Parte operativa.

A-5-3 3 byte - 6 ciclos JSR

PEPE	T0 - 1		
	T0 - 2		
	T0 - 3	Búsqueda	
	T0 - 4		
	T1 - 1	<PC> --> H, L	
	T1 - 2	JMP 1MAP	
1MAP	T1 - 3	INC PC	
	T1 - 4	<M> --> L1 (ADL) // # 1 --> H	
	T2 - 1	<S> --> L	WRITE
	T2 - 2	<PCH> --> M	WRITE
	T2 - 3	# 0 --> C(UAL)	WRITE
	T2 - 4	<S> - 1 --> S	WRITE
	T3 - 1	<S> --> L	WRITE
	T3 - 2	<PCL> --> M	WRITE
	T3 - 3	# 0 --> C(UAL)	WRITE
	T3 - 4	<S> - 1 --> S	WRITE
	T4 - 1	<PC> --> H, L	WRITE
	T4 - 2	<L1> --> PCL	WRITE
	T4 - 3	NOP	WRITE
	T4 - 4	<M> --> L1 (ADH)	WRITE

T5 - 1 <L1> --> PCH
 T5 - 2 NOP
 T5 - 3 NOP
 T5 - 4 JMP 2MAP

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa.

A-6-1 3 byte - 3 ciclos JMP Absoluto

PEPE T0 - 1
 T0 - 2
 T0 - 3 Búsqueda
 T0 - 4

T1 - 1 <PC> --> H,L
 T1 - 2 JMP 1MAP

1MAP T1 - 3 INC PC
 T1 - 4 <M> --> L1 (ADL)

T2 - 1 <PC> --> H,L
 T2 - 2 <L1> --> PCL
 T2 - 3 NOP
 T2 - 4 <M> --> L1 (ADH) // JMP 2MAP

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa.

A-6-2 3 byte - 5 ciclos JMP indirecto

PEPE T0 - 1
 T0 - 2
 T0 - 3 Búsqueda
 T0 - 4

T1 - 1 <PC> --> H,L
 T1 - 2 JMP 1MAP

1MAP T1 - 3 INC PC
 T1 - 4 <M> --> L1 (IAL)

T2 - 1 <PC> --> H,L
 T2 - 2 NOP
 T2 - 3 NOP
 T2 - 4 <M> --> L1 (IAH) // <L1> --> L,PCL

T3 - 1 <L1> --> H,PCH
 T3 - 2 NOP
 T3 - 3 INC PC


```

T3 - 4 <M> --> L1 (ADL)

T4 - 1 <PC> --> H,L
T4 - 2 <L1> --> PCL
T4 - 3 NOP
T4 - 4 <M> --> L1 (ADH) // JMP 2MAP

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa.

```

A-7 1 byte - 6 ciclos RTS

```

PEPE T0 - 1
      T0 - 2
      T0 - 3 Búsqueda
      T0 - 4

T1 - 1 <PC> --> H,L
T1 - 2 JMP 1MAP

1MAP T1 - 3 NOP
      T1 - 4 NOP

T2 - 1 NOP
T2 - 2 # 0 --> C(UAL)
T2 - 3 <S> + 1 --> S
T2 - 4 # 1 --> H

T3 - 1 <S> --> L
T3 - 2 # 0 --> C(UAL)
T3 - 3 <S> + 1 --> S
T3 - 4 <M> --> L1

T4 - 1 <S> --> L
T4 - 2 <L1> --> PCL
T4 - 3 NOP
T4 - 4 <M> --> L1

T5 - 1 <L1> --> PCH
T5 - 2 INC PC
T5 - 3 NOP
T5 - 4 JMP 2MAP

2MAP T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte operativa.

```

A-8 1 byte - 2, 3 ó 4 ciclos BRANCH

```
PEPE      T0 - 1
          T0 - 2
          T0 - 3      Búsqueda
          T0 - 4

          T1 - 1 <PC> --> H,L
          T1 - 2 JMP 1MAP

1MAP      T1 - 3 INC PC
          T1 - 4 <M> --> L1 (OFFSET) // IF (BCC, BCS, BEQ,
                                     ... Cond no se
                                     cumple) GOTO 2MAP

          T2 - 1 # 0 --> C(UAL)
          T2 - 2 <L1> + <PCL> --> L,PCL // <PCH> --> H
          T2 - 3 NOP
          T2 - 4 IF "C(UAL) = 0" THEN GO TO 2MAP

          T3 - 1 <PCH> + C(UAL) --> PCH
          T3 - 2 NOP
          T3 - 3 NOP
          T3 - 4 JMP 2MAP

2MAP      T0 - 1 <PC> --> H,L // JMP PEPE // Sin Parte ope-
                                     rativa.
```

APENDICE E

LISTADO DE MICROPROGRAMAS DEL INTERPRETE "J+1/2" PARA EL MICROPROCESADOR R6502

A-1 1 byte - 2 ciclos Sin referencia a memoria

NORMAL = 8 microinstrucciones

VERSION "J+1/2" = 3 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
1MAP      T1 - 3  <3MAP> --> MI // <HI> --> Camp Pag. MI //
           <PCL> --> LI // <PCH> --> HI

           T1 - 4  Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  <PCL> --> LI // <PCH> --> HI // JMP T1 - 4
```

A-2-1 2 byte - 2 ciclos (Inmediato)

NORMAL = 8 microinstrucciones

VERSION "J+1/2" = 3 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
1MAP      T1 - 3  <3MAP> --> MI // <HI> --> Camp Pag. MI //
           INC PC // <PCL> --> LI // <PCH> --> HI ya
           incrementado // <LI> --> LI2

           T1 - 4  <M> --> LBI,HBI // Leer MI // <MI> --> BMI
           (reg) // JMP 2MAP
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  <L>  --> L1 // INC PC // <PCL> --> LI  //
           <PCH> --> HI ya incrementado

           T0 - 3  Leer MI // <MI> --> BMI // JMP 2MAP
```

A-2-2 2 byte - 3 ciclos Página Cero

NORMAL = 12 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T2 - 2  <3MAP> --> MI // <HI> --> Camp Pag. MI //
           <H>  --> HB // <L>  --> LB
T2 - 3  <PCL> --> LI // <PCH> --> HI
T2 - 4  Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  INC PC // JMP T2 - 3
```

A-2-3 3 byte - 2 ciclos (Absoluto)

NORMAL = 16 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T3 - 2  <3MAP> --> MI // <HI> --> Camp Pag. MI //
           <L>  --> LB // <H>  --> HB
T3 - 3  <PCL> --> LI // <PCH> --> HI
T3 - 4  Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  INC PC
           T0 - 3  INC PC // <PCL> --> LI // <PCH> --> HI ya
           incrementado // JMP 2MAP
```

A-2-4 2 byte - 6 ciclos Indexado pág. cero, Indirecto

NORMAL = 24 microinstrucciones

VERSION "J+1/2" = 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <L> --> L1 // JMP T2 - 3

A-2-5 3 byte - 4 ó 5 ciclos Absoluto, X ó Y

NORMAL = 16 ó 20 microinstrucciones

VERSION "J+1/2" = 5 ó 9 microinstrucciones

Añadir horizontalmente a los microprogramas originales

1MAP T1 - 3 INC PC // <3MAP> --> MI // <HI> --> Campo
Pag. MI

T1 - 4 <M> --> LBI

T2 - 4 <M> --> HBI

T4 - 3 <PCL> --> LI // <PCH> --> HI

T4 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <L> + X ó Y --> L

T0 - 3 INC PC // IF "C(UAL) = 0" THEN GOTO T4 - 3

T0 - 4 <H> --> L1 // JMP T3 - 4

A-2-6 2 byte - 4 ciclos Página cero, X ó Y

NORMAL = 16 microinstrucciones
VERSION "J+1/2" = 8 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T3 - 3 <PCL> --> LI // <PCH> --> HI

T3 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <L> --> L1
 T0 - 3 JMP T2 - 4

A-2-7 2 byte - 5 ó 6 ciclos Indirecto, Y Pág. Cero

NORMAL = 20 ó 24 microinstrucciones
VERSION "J+1/2" = 12 ó 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T2 - 3

A-3-1 2 byte - 3 ciclos página cero STORE

NORMAL = 12 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T2 - 3 <PCL> --> LI // <PCH> --> HI

T2 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <A0X0Y> --> L1 // JMP T2 - 3 //
WRITE (en memoria principal)

A-3-2 3 byte - 4 ciclos (Absoluto) STORE

NORMAL = 16 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T3 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T3 - 3 <PCL> --> LI // <PCH> --> HI

T3 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <A0X0Y> --> L1 // WRITE
T0 - 3 INC PC // WRITE // <PCH> --> HI // <PCL> --
-> LI // JMP T3 - 4

A-3-3 2 byte - 6 ciclos Indexado X, Indirecto STORE

NORMAL = 24 microinstrucciones
VERSION "J+1/2" = 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // <L> --> LI // JMP T2 - 3 //
WRITE

A-3-4 3 byte - 5 ciclos Absoluto, X ó Y STORE

NORMAL = 20 microinstrucciones
VERSION "J+1/2" = 8 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T3 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T4 - 3 <PCL> --> LI // <PCH> --> HI

T4 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T3 - 4 // WRITE

A-3-5 2 byte - 4 ciclos Indexado pág. cero, X ó Y STORE

NORMAL = 16 microinstrucciones

VERSION "J+1/2" = 8 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T3 - 3 <PCL> --> LI // <PCH> --> HI

T3 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T2 - 3 // WRITE

A-3-6 2 byte - 6 ciclos Indirecto Indexado Y STORE

NORMAL = 24 microinstrucciones

VERSION "J+1/2" = 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T2 - 3 // WRITE

A-4-1 2 byte - 5 ciclos Página cero MODIFY

NORMAL = 20 microinstrucciones
VERSION "J+1/2" = 12 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
 <H> --> HB // <L> --> LB

T4 - 3 <PCL> --> LI // <PCH> --> HI

T4 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T2 - 3 // WRITE

A-4-2 3 byte - 6 ciclos Absoluto MODIFY

NORMAL = 24 microinstrucciones
VERSION "J+1/2" = 12 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T3 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
 <H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // WRITE
 T0 - 3 INC PC // JMP T3 - 4 // WRITE

A-4-3 2 byte - 6 ciclos Indexado pág. cero, X MODIFY

NORMAL = 24 microinstrucciones
VERSION "J+1/2" = 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T5 - 3 <PCL> --> LI // <PCH> --> HI

T5 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // JMP T2 - 3 // WRITE

A-4-4 3 byte - 7 ciclos Absoluto, X MODIFY

NORMAL = 28 microinstrucciones
VERSION "J+1/2" = 16 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T3 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T6 - 3 <PCL> --> LI // <PCH> --> HI

T6 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

3MAP T0 - 2 INC PC // WRITE
T0 - 3 INC PC // JMP T3 - 4 // WRITE

A-5-1 1 byte - 3 ciclos PUSH (PHP, PHA)

NORMAL = 12 microinstrucciones
VERSION "J+1/2" = 5 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T2 - 1 <ZMAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T2 - 3 <PCL> --> LI // <PCH> --> HI

T2 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

ZMAP T0 - 2 <S> --> L // JMP T2 - 2 // WRITE

A-5-2 1 byte - 4 ciclos PULL (PLA, PLP)

NORMAL = 16 microinstrucciones
VERSION "J+1/2" = 5 microinstrucciones

Añadir horizontalmente a los microprogramas originales

T3 - 2 <ZMAP> --> MI // <HI> --> Camp Pag. MI //
<H> --> HB // <L> --> LB

T3 - 3 <PCL> --> LI // <PCH> --> HI

T3 - 4 Leer MI // <MI> --> BMI (reg)

Microprograma que interpreta la Versión "J+1/2"

ZMAP T0 - 2 <S> + 1 --> S,L // JMP T3 - 2

A-5-3 3 byte - 6 ciclos JSR

NORMAL = 24 microinstrucciones
VERSION "J+1/2" = 11 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T5 - 2 <3MAP> --> MI // <HI> --> Camp Pag. MI //  
      <H> --> HB // <L> --> LB  
  
T5 - 3 <PCL> --> LI // <PCH> --> HI  
  
T5 - 4 Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP  T0 - 2 <L> --> LI (PCL) // <S> --> L // # 1 --> H  
      // WRITE  
T0 - 3 <PCH> --> M // WRITE  
T0 - 4 # 0 --> C(UAL) // WRITE  
T0 - 5 <S> - 1 --> S // WRITE  
  
T1 - 1 <S> --> L // WRITE  
T1 - 2 <PCL> --> M // WRITE  
T1 - 3 # 0 --> C(UAL) // WRITE  
T1 - 4 <S> - 1 --> S // WRITE // <HB> --> H //  
      <L1> --> PCL  
T2 - 1 <H> --> PCH // <PCL> --> LI  
T2 - 2 <PCH> --> HI // LEER MI // <MI> --> BMI //  
      JMP TO - 1 ( sin parte operativa )
```

A-6-1 3 byte - 3 ciclos JMP ABSOLUTO

NORMAL = 12 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
1MAP  T1 - 3 <3MAP> --> MI // <HI> --> Camp Pag. MI //  
      <LI> --> LI2  
T1 - 4 <M> --> LBI  
  
T2 - 3 <PCL> --> LI  
  
T2 - 4 <M> --> HBI, HI // Leer MI // <MI> --> BMI  
      (reg) // JMP TO - 1 (con parte operativa =  
      <L1> --> PCH, H)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  <L> --> PCL
          T0 - 3  <H> --> PCH // <PCL> --> LI
          T0 - 4  <PCH> --> HI // LEER MI // <MI> --> BMI //
                    JMP T0 - 1 ( sin parte operativa )
```

A-6-2 3 byte - 5 ciclos JMP Indirecto

NORMAL = 20 microinstrucciones

VERSION "J+1/2" = 8 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T3 - 2  <3MAP> --> MI // <HI> --> Camp Pag.  MI //
        <H> --> HB // <L> --> LB

T4 - 3  <PCL> --> LI

T4 - 4  <PCH> --> HI // Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP      T0 - 2  <H> --> PCH
          T0 - 3  <M> --> L1 (ADL) // INC PC // JMP T4 - 1
```

A-7 1 byte - 6 ciclos RTS

NORMAL = 24 microinstrucciones

VERSION "J+1/2" = 13 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T1 - 3  <3MAP> --> MI // <HI> --> Camp Pag.  MI

T5 - 3  <PCL> --> LI // <PCH> --> HI

T5 - 4  Leer MI // <MI> --> BMI (reg)
```

Microprograma que interpreta la Versión "J+1/2"

3MAP T3 - 1

A-8 1 byte - 2, 3 ó 4 ciclos **BRANCH**

NORMAL = 8, 12 ó 16 microinstrucciones
VERSION "J+1/2" = 4 microinstrucciones

Añadir horizontalmente a los microprogramas originales

```
T1 - 3  <3MAP> --> MI // <HI> --> Camp Pag.  MI //  
        IF "Cond." NO se cumple THEN (<PCH> --> HI  
        // <PCL> --> LI )  
T1 - 4  IF "Cond." NO se cumple THEN (LEER MI  //  
        <MI> --> BMI // JMP TO - 1)  
  
T2 - 3  IF "C(UAL) = 0" THEN (<PCL> --> LI //  
        <PCH> --> HI // <L> --> LB // <H> --> HB)  
  
T2 - 4  IF "C(UAL) = 0" THEN (Leer MI // <MI> -->  
        BMI (reg)// JMP TO - 1)  
T3 - 2  <H> --> HB // <L> --> LB  
T3 - 3  <PCH> --> HI // <PCL> --> LI  
T3 - 4  LEER MI // <MI> --> BMI // JMP TO - 1
```

Microprograma que interpreta la Versión "J+1/2"

```
3MAP            T0 - 2  IF "Cond." SI se cumple THEN (<H> --> PCH  
                  // <L> --> PCL) ELSE (INC PC) // JMP T3 -3
```

REFERENCIAS

- Ab74 A.M. Abd-Alla and D.C. Karlgaard. "Heuristic Synthesis of Microprogrammed Computer Architecture" IEEE Transactions on Computers C-23 (1974) 802-807.
- A176 F.E. Allen and J. Cocke "A Program Data Flow Analysis Procedure". Comm. Ass. Comput. Mach. Vol. 19, pp 137-147 (March 1976).
- Am83a G. Ambrozy et al. "Vertical Migration: Objectives, Supporting Tools and Experiences". Hungarian Academy of Sciences. Budapest 1983.
- Am83b G. Ambrozy "A Promising Application of Microprogramming : Vertical Migration". Third Symposium on Microcomputer and Microprocessor Applications. Budapest 1983. Pags 87-100.
- An72 F. Anceau. "A Microprogrammed System for Task Management" 1971 NATO Int. Adv. Summer School Proc. (Hermann, Paris 1972).
- Ay77a K.A. EL-Ayat "A Self-Tuning Microprogrammed Computer" Ph D dissertation, California Univ. Santa Barbara CA (1977).
- Ay77b K.A. El-Ayat and J.A. Howard "Algorithms for a Self-Tuning Microprogrammed Computer" Proc. 10th IEEE Annu. Workshop Microprogramming (1977) 85-91.

- Ba80 J.L. Baer "Computer Systems Architecture" Computer Science Press 1980.
- Bi85 V. Bitetto "Un Sistema di Sviluppo per Macchine Self-Tuning: Criteri di Progetto". Tesi di Laurea. Bari 1985.
- B183 M. De Blasi and G. Turco. "Real-Time Tuning of Computer Architecture" EUROMICRO 1983 pags 377-382.
- B184 M. De Blasi, A. Gentile, E. Luque y J. Sorribes "A Development System for Self Tuning Machines" EUROMICRO 1984 pags 145-148.
- B185 M. De Blasi, A. Gentile, E. Luque y A. Ripoll "Self-Tuning Machines" Euromicro Journal (1985) pags 195-201.
- Br76 G.E. Brown et al. "Operating System Enhancement through Microprogramming" SIGMICRO Newsletter 7 (1976) 28-33.
- Br77 G.E. Brown et al. "Operating System Enhancement through Firmware" SIGMICRO Newsletter 8 (3) (MICRO 10 Proc.) (1977) 119-133.
- Ch75 Y. Chu "High-Level Language Computer Architecture". Academic Press. New York 1975.
- Col85 A. Colella "Sistema di Specificazione e di Sviluppo di Microlinguaggi". Tesi di Laurea. Bari 1985
- Con85 G. Conforti "Un Sistema di Sviluppo per Macchine Self

- Tuning : Il Software di Sviluppo". Tesi di Laurea.
Bari 1985.
- De84 H.M. Deitel "An Introduction to Operating Systems"
Addison-Wesley Publishing (1984)
- Di87 T. Diez "Algoritmos de Selección en el Proceso de
Adaptación de la Arquitectura de un Ordenador" Tesis
Doctoral. Universidad Autónoma de Barcelona. 1987.
- Ha73 A. Hassitt et al. "Implementation of a High-Level
Language Machine" CACM 16 (4) (1973) 199-212.
- He81 J. Hemenway and R. Grappel "Understand the Newest Processor
to Avoid Future Shock." EDN. April 1981. pp 129-136.
- He86 P. Hernández "Simulador de Sistemas Digitales" Tesis
de Licenciatura. Universidad Autónoma de Barcelona
(1986).
- Ho84 B. Holtkamp y P. Wagner "An Algorithm for Selection of
Migration Candidates". The Seventeenth Annual Micro-
programming Workshop. 1984. Vol 15 pags 140-146.
- Ho85 B. Holtkamp " UNIX Requirements for Architectural
Support". Euromicro Journal 1985. pags 129-140.
- Lu77 E. Luque, L. Moreno y F. Tirado "A Multilingual High-
Level Processor" en B. Gilchrist (Ed) Information
Processing 77 (North-Holland, Amsterdam 1977).
- Lu80a E. Luque, A. Ripoll y J. Ruz "Dynamic Microprogramming
in Computer Architecture redefinition" EUROMICRO

Journal 6 (1980) 98-103.

Lu80b E. Luque y A. Ripoll "Tuning User Programs in a Microprogrammable environment" en G. Chroust and J.R. Muhlbacher (eds), Firmware, Microprogramming and Restructurable Hardware (North-Holland, Amsterdam 1980).

Lu80c E. Luque y A. Ripoll "Tuning Architecture via Microprogramming" Information Processing Letters 11 (1980) 102-109

Lu81a E. Luque y A. Ripoll "User Oriented Architecture" IEE Proceedings 128 Pt. E(4) (1981) 149-154.

Lu81b E. Luque y A. Ripoll "Microprogramming: A Tool for Vertical Migration" Microprocessing and Microprogramming 8 (1981) 219-228.

Lu83 E. Luque y A. Ripoll. "A Methodology for Vertical Migration". System Description Methodologies, Proceedings del IFIP TC2 Conference. Kecskemet (Hungria) 1983. pags 333-352.

Lu85 E. Luque, A. Ripoll, J. Sorribes y D. Rexachs "Adaptación Dinámica en Sistemas Microprogramados" Actas del VI Congreso de Informática Y Automática. Madrid 1985. pags 203-206.

Lu86 E. Luque, A. Ripoll, J. Sorribes y D. Rexachs "Improving Program Execution: Multiprocessing and Dynamic Migration". in Microprocessor Advanced

- Architectures and Design Methodologies (de Blasi & Luque Eds). Edizioni Fratelli Laterza. Bari 1986.
- Lu87a E. Luque, J. Sorribes y A. Ripoll "Coprocessor for Real-Time Dynamic Vertical Migration". EUROMICRO. Microprocessing and Microprogramming 20. Venecia (1987) pags 197-202.
- Lu87b E. Luque, A. Ripoll y T. Diez "Vertical Migration: An Experimental Study for the Candidates Selection Problem" IEE Proceedings-E Computers and Digital Techniques. Por aparecer.
- Mi85 V. Milutinovic et alt. "Advanced Microprocessors and High-Level Language Computer Architecture" IEEE Computer Society Press. Los Angeles 1986.
- My81 G. J. Myers "Advances in Computer Architecture" Segunda edición. J. Wiley & Sons (Ed). 1981.
- Na84 G. Natilla "Un Sistema di Sviluppo per Macchine Microprogrammabili". Tesi di Laurea. Bari 1984.
- No84 P.S. Noviello "Progetto del Livello di Microprogrammazione di un Emulatore". Tesi di Laurea. Bari 1984.
- Ra75 T.G. Rauscher "Dynamic Problem Oriented Redefinition of Computer Architecture Via microprogramming" PH. D. dissertation, Department of Computer Science, Univ. Maryland (1975).
- Ra76 T.G. Rauscher and A.K. Agrawala "Developing Applica-

- tion Oriented Computer Architectures on General Purposes Microprogrammable Machines" NCC Proc., Montvale (AFIPS Press, 1976) 715-720.
- Ra78 T.G. Rauscher and A.K. Agrawala "Dynamic Problem Oriented Redefinition of Computer Architecture Via Microprogramming" IEEE Transaction Computers C-27 (1978) 1006-1014.
- Ri80 A. Ripoll "Adaptación de la Arquitectura en Sistemas Microprogramables" Tesis Doctoral. Universidad Autónoma de Barcelona. 1980.
- Ro78 R6500 "Microcomputer System Hardware Manual" Rockwell International. 1978.
- Sh83 H. Shin y M. Malek "Identification of Microprogrammable Loops for Problem Oriented Architecture Synthesis". Proceedings of 16th Annual Microprogramming Workshop October 1983.
- Sm82 A.J. Smith. "Cache Memories". Computing Surveys. Vol. 14. N.3. pp 473-530. Sept. 1982.
- So82 J. Sorribes "Sistema para el Desarrollo de Microprogramas" Tesis de Licenciatura. Universidad Autónoma de Barcelona. 1982.
- St78 J. Stockenberg and A. van Dam "Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems" Computer 11 (5) (1978) 35-50.

We67 H. Weber "A Microprogrammed Implementation of EULER on
IBM System 360 Model 30" CACM 10 (1967) 549-558.

Wi69 N. Wirth "On Multiprogramming, Machine Coding and
Computer Organization" CACM 12 (9) (1969).

Wi72 W.T. Wilner "Design of the B-1700" Proc. FJCC (AFIPS
Press, 1972) 489-497.