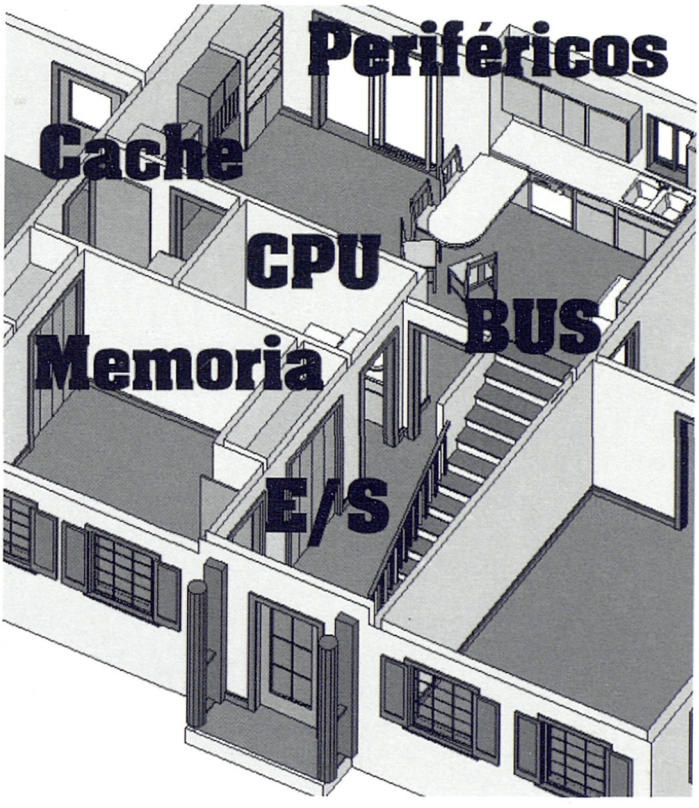
RAFAEL ASENJO PLAZA
ELADIO GUTIÉRREZ CARRASCO
JULIÁN RAMOS CÓZAR

FUNDAMENTOS DE LOS COMPUTADORES



UNIVERSIDAD DE MÁLAGA / MANUALES

Departamento de Arquitectura de Computadores

Universidad de Málaga

Fundamentos de los Computadores

Rafael Asenjo Plaza

Eladio Gutierrez Carrasco

Julián Ramos Cózar

Málaga, 2001

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

UNIVERSIDAD DE MÁLAGA

Publicaciones y
Divulgación Científica

Quinta edición, febrero 2006

© Los autores
© Publicaciones y Divulgación Científica. Universidad de Málaga.

Diseño de Ia colección: J. M. Mercado

I.S.B.N.: 84-7496-855-0
Depósito Legal: MA-141/2006

Imprime: Imagraf Impresores, S.A. Tel.: 952 32 85 97

Esta obra está bajo una licencia de Creative Commons
Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:
http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode
Cualquier parte de esta obra se puede reproducir sin autorización
pero con el reconocimiento y atribución de los autores.
No se puede hacer uso comercial de la obra y no se puede alterar,
transformar o hacer obras derivadas.

Aviso

- Este material ha sido preparado por:

  Rafael Asenjo Plaza
  Eladio Gutierrez Carrasco
  Julián Ramos Cózar
  Dept. de Arquitectura de Computadores. Universidad de Málaga.

- Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivar 4.0 Internacional.

- Debe dar crédito en la obra en la forma especificada por el autor o licenciante.

- El licenciante permite copiar, distribuir y comunicar publicamente la obra. A cambio, esta obra no puede ser utilizada con fines comerciales – a menos que se obtenga el permiso expreso del licenciante.

- El licenciante permite copiar, distribuir, transmitir y comunicar públicamente solamente copias inalteradas de la obra – no obras derivadas basadas en ella.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Índice

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Índice de figuras

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Índice de tablas

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Prefacio

Parece que ya deja de sorprender la creciente omnipresencia de los ordenadores en todos los ámbitos de la sociedad. A nosotros nos sigue impresionando la rapidísima evolución de los sistemas basados en computador, su creciente potencia de cálculo capaz de resolver cada vez problemas de mayor complejidad y su capacidad de simplificar y reducir el tiempo necesario para realizar muchas tareas cotidianas. Pues bien, los fundamentos, conceptos y modos de operación de estas máquinas tan comunes hoy en día, son los que tratamos de introducir y desentrañar en este texto. O con otras palabras, este libro está orientado a aquellas personas que alguna vez se han preguntado "¿Cómo es posible que los transistores y puertas lógicas que hay dentro de mi ordenador me permitan editar un archivo o ejecutar un programa que he escrito en Modula o en C?", pregunta, que por otro lado, esperamos se hayan planteado todos nuestros alumnos de asignaturas de introducción a los computadores (Fundamentos de los Computadores, Introducción a los Computadores, ...).

Más particularmente, los contenidos de este libro nacen de los apuntes desarrollados para la asignatura Fundamentos de los Computadores. Esta asignatura se imparte en el primer curso de la ETSI de Telecomunicación de Málaga durante el segundo cuatrimestre, a razón de dos horas por semana. A esas alturas del año, los alumnos ya han cursado la asignatura de Electrónica Digital 1 y Elementos de Programación, impartidas en el primer cuatrimestre. Por tanto, en este libro queremos cubrir el desnivel semántico que existe en un sistema computador entre los contenidos de estas dos asignaturas previas (electrónica digital y lenguajes de alto nivel), contemplando el control microprogramado y cableado, el lenguaje ensamblador y los sistemas operativos, según desglosamos a continuación por temas.

El tema 1 introduce los primeros conceptos básicos y una descripción inicial de la arquitectura de von Neumann. El tema 2 describe los convenios comúnmente utilizados para representar números, caracteres e instrucciones en el computador. A partir del tema 3 profundizamos en la arquitectura de computadores siguiendo un esquema estructural, en el que el computador se compone del procesador (tema 3), el cual engloba la sección de control (tema

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

4) y de procesamiento (tema 5), jerarquía de memoria (tema 6) y unidad de Entrada/Salida (tema 7). Por último, el tema 8 describe como los Sistemas Operativos permiten gestionar toda la arquitectura, dando una visión global del computador. En esta segunda edición se ha incluido material adicional en algunos temas y varios apéndices nuevos para cubrir también los contenidos de la asignatura que con el mismo nombre se imparte en segundo curso de la ingeniería técnica de telecomunicación.

Hemos creído conveniente recoger en un único libro estos contenidos en el orden y con la extensión con que se imparten en esta asignatura. De esta forma, se facilita al alumno de primero el seguimiento de la asignatura, ahorrándole en gran medida el tiempo necesario para consultar la extensa literatura que cubre este campo e interpretar los distintos puntos de vista con que se tratan los mismos conceptos en cada uno de los libros. De cualquier modo, el alumno interesado siempre puede consultar la bibliografía propuesta al final del libro cuando su curiosidad le demande una mayor profundidad en sus conocimientos de arquitectura.

Por otro lado, hemos omitido en el libro la referencia a arquitecturas concretas que pueden quedar obsoletas en poco tiempo. Por ejemplo, la arquitectura PC (la más familiar al alumno) del curso en que empezamos a impartir esta asignatura (94/95), contenía probablemente un Pentium a 75Mhz con 3.1 millones de transistores, cache L1 de 8Kbytes de instrucciones y otros tantos de datos, 16Mbytes de RAM FPM en módulos SIMM, 800 Mbytes de disco duro IDE-ATA, por citar algunos números y estándares. Han pasado seis años y para el curso que llaman del nuevo milenio ya disponemos de un Pentium4 con 42 millones de transistores, cache de más 256K, 128Mbytes de SDRAM, DDR o RDRAM en módulos DIMM o RIMM y más de 20Gbytes de disco duro Ultra-DMA ATA-100. Aunque pensamos actualizar este texto cada dos años, hemos preferido reflejar en el libro sólo los aspectos de la arquitectura más invariantes en el tiempo (arquitectura del conjunto de instrucciones, estrategias de control, algoritmos aritméticos, entrelazamiento de memoria, cache, interrupciones, DMA, etc.) que serán completados con ejemplos de arquitecturas actuales durante cada clase.

Evidentemente, otra fuente de información actualizada es la WEB. Por ello, mantenemos desde el año 96 páginas de las asignaturas:

http://www.ac.uma.es/academia/educacion/cursos/FundCompTel/

http://www.ac.uma.es/academia/educacion/cursos/FundCompST/

con apuntes, formulario de consulta/tutoría on-line, enlaces a páginas de in-

terés para completar conocimientos de arquitectura y otras noticias, como notas de los exámenes, aportaciones de alumnos, etc., que esperamos sea de utilidad y también motive al alumno a familiarizarse con su ordenador personal.

En último lugar, aunque no menos importante, queremos agradecer la colaboración de los miembros del Departamento de Arquitectura de Computadores, en particular a José María González, por sus aportaciones y revisiones, a Oscar Plata, por administrar el servidor web del departamento, y a Emilio L. Zapata que fue quien nos enseñó arquitectura de computadores cuando éramos nosotros los alumnos. Tampoco olvidamos las aportaciones de nuestros alumnos, que nos han señalado las erratas y nos han ayudado a detectar los aspectos que necesitaban una explicación más detallada mediante sus cuestiones y comentarios en clase. A ellos dirigimos y dedicamos este texto.

Los Autores

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

1 Introducción a los Computadores

OBJETIVOS

- Describir las características básicas de los computadores del tipo von Neumann
- Proporcionar una perspectiva histórica

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

1.1. NATURALEZA DE LOS COMPUTADORES

Empecemos esta introducción a la arquitectura de computadores recordando dos definiciones básicas:

Computador: Sistema (conjunto de elementos interrelacionados) al que se le suministran datos y nos aporta unos resultados acordes con la resolución de un determinado problema. Luego el objetivo del computador es resolver problemas. Para ello ejecuta programas de forma rápida, eficaz y cómoda.

Programa: Secuencia de instrucciones que resuelven el problema mencionado en la definición anterior.

Para resolver problemas mediante programas también se utilizan algunas herramientas:

- Sistema Operativo (S.O.): facilitan la gestión de los recursos.
- Compiladores de lenguajes de alto nivel.

1.2. ANTECEDENTES HISTÓRICOS

Desde tiempos muy antiguos el hombre ha necesitado hacer cálculos. Para ello se ha buscado máquinas que lo ayudasen, como el ábaco, de origen oriental, perfeccionado por los griegos; las Hileras de John Napier, que facilitaban la multiplicación y la división; las Reglas de Cálculo, para calcular funciones trigonométricas, exponenciales y logaritmos; etc.

1.2.1. Progresos mecánicos

A partir del siglo XVII se realizan progresos mecánicos en los sistemas de cálculo. Aparecen las calculadoras mecánicas por varios motivos: investigación, y principalmente económicos, para el cálculo de tablas astronómicas (útiles en navegación) y para recaudar impuestos.

- En 1623, W. Shickard (astrónomo amigo de Kepler) construye la primera calculadora, basada en ruedas dentadas y capaz de sumar y multiplicar.
- En 1642, B. Pascal, a los 19 años de edad construye una máquina que suma y resta. Quería ayudar a su padre que era recaudador de impuestos.
- En 1671, Leibnitz mejora la máquina de Pascal, diseñando la multiplicación por iteraciones. Ahora la nueva máquina suma, resta, multiplica y divide. Realiza algunas aportaciones importantes:
  - consigue cierto avance en la mecánica de precisión

  - intuye que la máquina trabajará mejor con números binarios. (Esta idea la retoma Boole para elaborar la Lógica de Boole en 1854, y posteriormente Claude Shannon para sentar las bases de la Conmutación en 1937).

- En 1820, Charles Babbage, realiza el progreso más importante: construye en parte la máquina analítica. Será el primer computador digital de propósito general. Características de esta máquina:
  - trabaja con aritmética de 50 dígitos decimales
  - consigue velocidades de un segundo para sumar y restar y un minuto para multiplicar y dividir
  - trabaja con tarjetas perforadas para indicar las operaciones a realizar y las variables (programa externo).
  - la estructura de su máquina se aprecia en la figura 1.1.

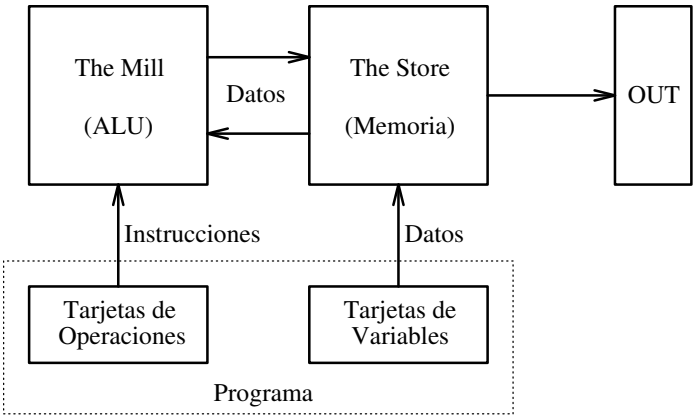


Figura 1.1: Máquina de Charles Babbage

- En 1880, Herman Holleritz, crea una máquina capaz de hacer estadísticas sobre censos.

1.2.2. Progresos electromecánicos. Relés

- En 1914, Leonardo Torres Quevedo, con tecnología electromecánica diseña los esquemas de una calculadora digital de propósito general. Emplea representación en coma flotante.
- En 1938, G. Stibitz, construye una máquina denominada Model I que

trabaja con números complejos, para realizar cálculos balísticos. Fue el primer sistema con terminales remotos. Uno de sus sucesores, el Model V, consta de dos procesadores y un total de 9000 relés. El Model VI es gestionado por uno de los primeros sistemas operativos (proceso por lotes -batch-).

- En 1941, K. Zuse (Alemán) construye las máquinas Z1, Z2 y Z3. Esta última es de propósito general con 2600 relés. El programa sigue siendo externo.
- En 1942, Atanasoff, físico de la universidad de Iowa, construyó una máquina de propósito general con tubos de vacío (conmutador electrónico). Realmente construyó un prototipo de máquina para el cálculo de ecuaciones diferenciales. Por culpa de la guerra no terminó la máquina, pero es considerado el inventor del ordenador.

1.2.3. Progresos electrónicos

- En 1943, Mauchly y Eckert construyeron el ENIAC (Electronic Numeric Integrator and Calculator). Fue considerado el primer computador digital electrónico hasta el 1973. Su máquina estaba orientada a resolver ecuaciones diferenciales y tablas balísticas.

  Características:

  - 18000 tubos de vacío
  - Consume 150 Kw
  - Dimensiones: 30m de largo por 2.5m de alto
  - Tenía 20 registros de 10 dígitos decimales
  - Trabajaba con aritmética en punto fijo
  - Velocidad: 200 μs para sumar, 2.8 ms para multiplicar y 6 ms para dividir
  - Introducción de datos mediante tarjetas perforadas
  - La secuencia de operaciones se establecía mediante un conexionado manual entre elementos, y por las posiciones de ciertos interruptores. Introducir o modificar programas era muy tedioso.
- En 1945, John von Neumann entra en el proyecto ENIAC. Será el encargado de facilitar la programación del ENIAC. Conocía bien el trabajo de Babbage y su máquina analítica. Aporta una idea innovadora: Almacenar en una memoria rápida datos e instrucciones, y así modificar el programa consistiría sencillamente en modificar las instrucciones almacenadas en memoria.

  Diseña el EDVAC (Electronic Discrete Variable Automatic Compu-

ter). Esta máquina tiene más memoria pero es más lenta. Utiliza 44 bits en punto fijo. Las instrucciones son de cuatro direcciones y tiene dos unidades aritméticas. Sin embargo no fue la primera máquina con programa almacenado.

- En 1946, Maurice Wilkes, de la universidad de Cambridge comienza el EDSAC (Electronic Delay Storage Automatic Calculator). Acaba en 1949 (tres años antes que el EDVAC).
- En 1946 von Neumann entra en el Institute for Advanced Study (IAS de Princeton). Con subvención militar construye el IAS. Algunas de sus características son:
  - Dimensiones 2.4×2.4×0.6 m
  - Memoria de 1024 palabras de 40 bits

1.3. ARQUITECTURA DE VON NEUMANN

La mayoría de los computadores se han construido siguiendo la arquitectura de la máquina de von Neumann.

1. ESTRUCTURA (figura 1.2):

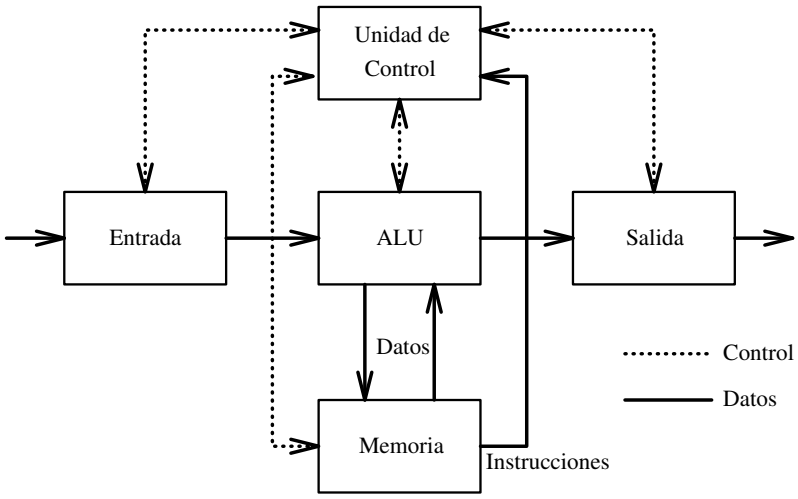


Figura 1.2: Máquina de von Neumann

a) Entrada.- Unidad que transmite instrucciones y datos del exterior a la memoria (pasando por la ALU).

b) Memoria.- Unidad que almacena instrucciones y datos, así como los resultados parciales y finales de los programas.
c) A.L.U.- Unidad que realiza las operaciones aritmético-lógicas (suma, multiplicación, OR, con datos de memoria). Tiene registros para almacenar operandos: RALU.
d) Unidad de Control.- Interpreta las instrucciones y coordina el resto del sistema.
e) Salida.- Transmite los resultados al exterior.

2. ARQUITECTURA:

a) Una sola memoria direccionada secuencialmente. La memoria tiene una estructura lineal (vector de palabras direccionables). Es el cuello de botella del sistema.
b) No hay distinción explícita entre instrucciones y datos. Los datos no tienen ningún significado explícito.
c) El procesamiento es totalmente secuencial. Las fases de ejecución de las instrucciones son secuenciales (sólo hay una unidad de control).
d) Hay instrucciones que permiten la ruptura de secuencia (es decir, se permiten los saltos en el programa).

3. FUNCIONAMIENTO:

El objetivo del procesador es ejecutar instrucciones. Las instrucciones son órdenes elementales (instrucciones máquina). Partimos de un programa máquina (conjunto de instrucciones máquina) en memoria y veremos cómo se ejecuta. Como las instrucciones están en memoria accedemos a ellas mediante direcciones. Una vez que conocemos la dirección de la siguiente instrucción a ejecutar, las fases son las siguientes:

a) Búsqueda de la instrucción (fetch).
b) Decodificación.
c) Cálculo de la dirección de los operandos.
d) Búsqueda de los operandos.
e) Ejecución (realiza la operación y almacena el resultado).

Cada instrucción máquina debe especificar:

a) La operación a realizar en el Código de Operación (C.O.)
b) Información para calcular las direcciones de los operandos y dónde se guarda el resultado (Modos de Direccionamiento).
c) Información de la dirección de la próxima instrucción a ejecutar.

4. DEFINICIONES:

   - **RALU:** ALU + Registros.
   - **PROCESADOR:** U.C. + RALU + lógica adicional para interconectar todos los elementos.
   - **MICROPROCESADOR:** procesador integrado en una pastilla de silicio (chip).

1.4. GENERACIONES DE COMPUTADORES

Se presentará la evolución de los computadores agrupándolos en generaciones según la tecnología de fabricación.

- Primera Generación: válvulas.
- Segunda Generación: Transistores (TRT) discretos.
- Tercera Generación: *Small Scale Integration* o SSI (<100 TRT's) - Medium SI o MSI (<3000 TRT's).
- Cuarta Generación: Large SI o LSI (<30.000 TRT's) - Very Large SI o VLSI (cientos de miles).
- Quinta Generación: Inteligencia Artificial. Lenguajes de programación lógicos (Japón).
- Sexta Generación: Computadores paralelos, vectoriales y superescalares.

1. GENERACIÓN 1.ª (1938 - 1952)
   - Tecnología: válvulas de vacío. Eran máquinas voluminosas, de alto consumo, caras y de vida limitada.
   - Máquinas: ENIAC, EDVAC, EDSAC, UNIVAC I, IAS, y las comerciales IBM 701, 704, 709.
   - Avances del equipo físico: en la memoria se pasa de registros de válvulas a núcleos de ferrita; en la memoria secundaria, de tarjetas y cintas perforadas a tambores y cintas magnéticas. Además se introduce el control de interrupciones.
   - Avances del equipo lógico: utilización de aritmética binaria, programación en ensamblador (para ayudar al programador).
2. GENERACIÓN 2.ª (1953 - 1962)
   - Tecnología: en 1948 se inventó el transistor en los laboratorios de la Bell. Pero hasta 1954 no construyeron el TRADIC en la Bell, que fue el primer computador transistorizado. Las ventajas del transistor son que es más pequeño, el consumo es menor y más barato que las válvulas. Con lo cual los computadores se hacen más asequibles.

- Máquinas: UNIVAC 1107, BURROUGH D-805, PDP-5 de DEC, y las científicas IBM 7070, 7090, 7094.
- Avances del equipo físico: se consolidan las memorias de ferrita. Aparecen los canales de E/S.
- Avances del equipo lógico: aparecen los lenguajes de alto nivel (FORTRAN, COBOL, ALGOL, PL1). Se impone el procesamiento tipo batch o por lotes: ejecución automática y secuencial de los programas de usuario, uno a uno.

3. GENERACIÓN 3.ª (1963 - 1971)
   - Tecnología: se integran los transistores y aparecen los Circuitos Integrados (C.I.): SSI, MSI.
   - Máquinas: IBM 360. Aparecen las "Familias de Computadores": computadores de distinta potencia y precio pero con la misma arquitectura y totalmente compatibles. Se produce una explosión de los mini-computadores: recursos más limitados pero muy asequibles (PDP-8, PDP-11).
   - Avances del equipo físico: tarjetas de circuito impreso (PCB); memorias electrónicas sustituyen a las de ferrita; aparecen las memorias cache; la CPU está basada en registros de propósito general.
   - Avances del equipo lógico: nuevos lenguajes de alto nivel (BASIC, PASCAL); gran avance en el S.O.; aparece la multiprogramación.
4. GENERACIÓN 4.ª (1972 - 1987)
   - Tecnología: se incrementa la escala de integración (LSI, VLSI). Se puede integrar en un chip todo un procesador. Los computadores con microprocesador se llamaron microcomputadores.
   - Máquinas: se pueden distinguir 4 fases:
     a) 1.ª fase (1971 - 74): microprocesador de 4 bits, como el INTEL 4004 con 2300 TRT's y LSI.
     b) 2.ª fase (1974 - 76): microprocesador de 8 bits, como el 8080 de INTEL con 5000 TRT's, el 6800 de Motorola con 6000 TRT's, o el Z80 de Zilog.
     c) 3.ª fase (1976 - 80): microprocesador de 16 bits, como el 8086 de INTEL (con 29000 TRT's), el Z8000 de Zilog o el 68000 de Motorola.
     d) 4.ª fase (1980 - 87): microprocesador de 32 bits, como el 80286 (con 134000 TRT's), el 80386 (con 275000 TRT's) de INTEL, o el 68020 y el 68030 de Motorola.
   - Avances del equipo físico: más integración de las memorias; los dis-

cos duros tienen más capacidad; aparecen los coprocesadores para el tratamiento en punto flotante FPU y los gestores de memoria o MMU.

- Avances del equipo lógico: se normaliza el uso de la memoria virtual; los S.O. permiten la multitarea y el multiproceso; se producen avances en los lenguajes de alto nivel.

5. GENERACIÓN 5.ª (desde 1988 hasta hoy)
   - Tecnología: los modernos CI's son hoy en día capaces de integrar más de 1 millón de TRT's.
   - Máquinas: Distinguimos tres apartados:
     a) Procesadores de altas prestaciones. Las familias de procesadores siguen evolucionando: **INTEL**[1]: 486 (1.2, 25-100MHz), Pentium (3.1, 60-233MHz), Pentium Pro (5.5, 150-200MHz), PentiumII (7.5, 233-450MHz), PentiumIII (44, 450MHz-1.4GMHz), Pentium M (144, 1.2-2GHz), Pentium 4 (108, 2-3.8GHz), y otros (Pentium D, Pentium EE, Itanium II ...); **AMD**: K5, K6, K6 3D, K6+ 3D, Athlon (K7), Athlon 64, Opteron; **Procesadores RISC**: Power 4 y 5, Alpha 21064 - 21364, UltraSPARC, R10000 - R16000, etc.
     b) Supercomputadores de alta velocidad. Aparecen porque se buscaba más velocidad. Se emplean fundamentalmente para procesado de imagen, control de robots, simulación, etc. Para aumentar la velocidad se incrementa el HW, se utilizan más microprocesadores, dando lugar a sistemas multiprocesador y al paralelismo, y empleando la "segmentación" y las máquinas vectoriales.
     c) Computadores de funciones inteligentes. Inicialmente la computación se realizaba sobre datos. Posteriormente se realizó sobre información (conjunto de datos relacionados). Pero ahora, lo que se pretende con este tipo de máquinas es la computación sobre el conocimiento (es decir, cuando a la información se le da valor semántico). Se quieren diseñar computadores capaces de tomar decisiones inteligentes en función de un conjunto de conocimientos (base de conocimientos). Aparecen los sistemas expertos, como el MYCIN, que se emplea para diagnósticos de enfermedades infecciosas a partir de determinados síntomas.

[1] Los dos números entre parantesis indican los millones de transistores y el rango de frecuencias de trabajo de esos procesadores.

1.5. NIVELES DE DESCRIPCIÓN

Se define un sistema como una combinación de elementos que actúa conjuntamente y cumplen con un determinado objetivo o función. Podemos representarlo según se ve en la figura 1.3, donde $z = F(x)$, y llamamos a F función de transferencia.

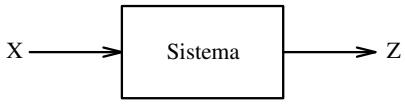


Figura 1.3: Representación de un Sistema

El comportamiento de entrada/salida es el conjunto de relaciones que el sistema establece entre la entrada y la salida. Podemos hablar de sistema físico, sistema digital, sistema analógico, sistema de telecomunicación, ...

1.5.1. Especificación e implementación de un sistema

Podemos distinguir entre:

- La especificación de un sistema que consiste en la descripción funcional de lo que hace el sistema a alto nivel.
- La implementación del sistema que es la realización del sistema, es decir, la construcción del sistema satisfaciéndose requisitos tales como prestaciones y coste.

El paso de la especificación a la realización (implementación) se consigue mediante la síntesis o diseño. El paso contrario, como vemos en la figura 1.4, es el análisis.

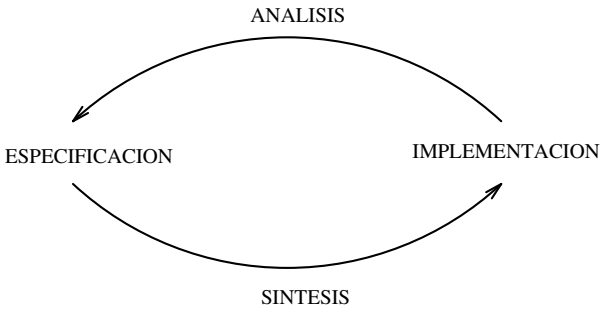


Figura 1.4: Especificación e Implementación

Si nos centramos en un sistema particular como puede ser un sistema de computadores, lo podemos describir también siguiendo esas dos alternativas:

1. Dando una descripción funcional del microprocesador. Esta descripción es la que viene en las hojas de datos comerciales, que cuentan cómo funciona, qué instrucciones máquina puede ejecutar (ISP), cómo se gestionan las interrupciones, etc.
2. Dando una descripción lógica del microprocesador. Consiste en describir el circuito a nivel de puertas. Ésta es una información que suele estar reservada por los fabricantes. Es en definitiva la implementación.

Conociendo la descripción funcional, se sabe qué hace el computador y cómo funciona, pero no cómo está construido (implementado). Por el contrario, conociendo la descripción lógica se sabe cómo está hecho, pero es muy difícil determinar su funcionamiento y el modo de operación a partir de esa información.

Entre estos dos niveles de descripción extremos de un microprocesador, se puede considerar otro, el denominado nivel de transferencia de registros, RTL. Mediante este nivel de descripción intermedio, se especifican los registros y cómo se transfiere información entre ellos: cómo pasa la información por la ALU, cómo se guarda en memoria, etc. De esta forma, sabremos un poco de la implementación (a alto nivel) y, por otro lado, será más sencillo determinar el funcionamiento del computador a partir de la descripción RTL. Es el nivel que básicamente utilizaremos en esta asignatura.

SINOPSIS

Resumiendo, este tema ha servido tanto para introducir y repasar algunos conceptos básicos, como para recorrer las etapas cubiertas en la evolución de los sistemas de cálculo y computación. También hemos hecho especial hincapié en la arquitectura de von Neumann debido a que es la base sobre la que se han construido los computadores modernos. Como apunte final, subrayar la vertiginosa evolución del campo del que trata esta asignatura: por poner sólo un ejemplo, comparando el ENIAC con un PentiumIII a 500MHz, el tiempo necesario para realizar una suma se ha reducido en 5 ordenes de magnitud en apenas 50 años.

2 Representación de la Información

OBJETIVOS

- Estudiar los modos de representación de enteros y flotantes
- Entender el formato de las instrucciones y los modos de direccionamiento

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

2.1. CLASIFICACIÓN DE LA INFORMACIÓN

El ordenador para resolver problemas ejecuta programas en los que se procesa información. Para representar información normalmente se utiliza un alfabeto de símbolos (el alfabeto de las letras, los dígitos, las notas musicales...). El alfabeto del ordenador solo tiene dos símbolos: 0 y 1.

Las razones de que sólo se utilicen dos símbolos son:

1. Es más fácil trabajar con dispositivos de dos estados, que con dispositivos de diez estados. Por ejemplo: los relés, las válvulas y los TRT's. Aunque los TRT's tienen 4 modos de funcionamiento, tan sólo 2 de ellos (corte y saturación) son los que se emplean en electrónica digital. Un estado representa el 0 y el otro el 1. Además, la conmutación entre estos dos estados ha de ser lo más rápida posible (en ruedas dentadas con 10 estados estas conmutaciones eran muy lentas).
2. El sistema binario tiene casi la mejor eficiencia de almacenamiento de la información.

   Sea X codificado como un vector de dígitos

   $$x_n, x_{n-1}, ..., x_1, x_0, x_{-1}, ..., x_{-m},$$

   teniendo en cuenta que:

   a) b es la base o radix.
   b) El rango de cada dígito va desde 0 a b-1.
   c) $|x| = \sum_{-m}^{n} x_i b^i$.
   d) Con n dígitos, se pueden representar $N = b^n$ números posibles.
   e) Sea la magnitud $E = nb = (\ln N / \ln b) \cdot b$, que expresa el coste asociado a almacenar b^n valores distintos usando una base b y un vector de n dígitos.

   El valor que minimiza E, es decir, la base que fuerza $dE/db = 0$ y $d^2E/d^2b > 0$, es $b = e$. Como la base ha de ser un número entero y $2 < e < 3$, la base de los computadores digitales es 2.

Los inconvenientes de esta elección son básicamente:

1. Cada vez que queremos comunicar algo al computador es necesario realizar una conversión al código binario, y viceversa.
2. A veces los números binarios tienen una longitud excesiva, y las secuencias pueden ser muy largas y engorrosas.

En un computador la información se representa por medio de secuencias binarias, las cuales se organizan en "palabras". Se define una "palabra" como

la cantidad de información codificada mediante n dígitos binarios, donde n es determinado atendiendo a diversas consideraciones (rango de representación, coste HW, etc...). Los valores típicos de n son:

- n=4 ⇒ *nibble*
- n=8 ⇒ *byte*
- n=16 ⇒ *word*
- n=32 ⇒ *double word*

Como vemos, normalmente se utilizan palabras con n potencia de 2. Los computadores actuales tienen tamaños de palabra múltiplo de 8: 8, 16, 32 y 64 bits.

Pues bien, como hemos dicho, vamos a codificar la información mediante cadenas de 1's y 0's. Pero la codificación se hará de distinta forma en función de qué tipo de información vayamos a codificar. Por tanto, clasificaremos los tipos de información que necesitamos representar en el computador según el siguiente esquema:

- Instrucciones (Información de Control)
- Datos
  - No Numéricos (caracteres alfanuméricos)
  - Numéricos
    - *Enteros*
      - Posicionales
      - No Posicionales
    - *Reales*
      - Punto Fijo
      - Punto Flotante

y en los siguientes apartados iremos viendo, para cada uno de los tipos de información, qué estrategia de codificación utilizamos normalmente.

Pero antes de entrar en el estudio de la representación de los datos y de las instrucciones, nos vamos a plantear una cuestión que se presenta inevitablemente, debido a que tanto los datos como las instrucciones se representan mediante cadenas de unos y ceros: desde que von Neumann almacenó tanto datos como instrucciones en memoria, estos dos tipos de información son **indistinguibles** entre sí (es decir, si elegimos una palabra aleatoriamente de la memoria no podemos decir si es un dato o una instrucción). Algunos diseñadores argumentaban que se debería añadir una etiqueta (secuencia de bits especial) para distinguir los datos de las instrucciones y evitar cosas como intentar ejecutar un dato. Pero añadir etiquetas incrementa el tamaño de la memoria, el coste HW y realmente es innecesario. Si el procesador esta correctamente diseñado y el programa a ejecutar se inicializa adecuadamente y está bien programado, un registro, llamado Contador de Programa (PC), per-

mite identificar qué posiciones de memoria contienen una instrucción y cuáles un dato.

2.2. REPRESENTACIÓN DE LOS DATOS

2.2.1. Datos Numéricos

Los factores a tener en cuenta para representar números son:

1. el tipo de los números a representar (enteros, reales, etc)
2. el rango de números representables
3. la precisión del dato numérico (n.º de números para un rango dado)
4. el coste HW requerido para almacenar y procesar los números a representar.

2.2.2. Enteros

1. Posicionales, ponderados o pesados.

   Son los utilizados por la gran mayoría de computadores. Un número se representa como un vector o secuencia de dígitos, donde cada uno tiene un peso de acuerdo con la posición que ocupa. Si la base de representación es b, el rango de cada dígito va de 0 a $b-1$. Como hemos dicho anteriormente, en los sistemas de computadores se emplea la base $b = 2$.

   O sea, si el número x se codifica mediante el siguiente vector de dígitos:

   $$x_n, x_{n-1}, ..., x_1, x_0, x_{-1}, ..., x_{-m}$$

   su valor es

   $$|x| = \sum_{-m}^{n} x_i b^i \tag{2.1}$$

   En principio hablaremos, en esta sección, únicamente de números enteros, por lo tanto no encontraremos potencias negativas en el sumatorio de la última ecuación (es decir el rango de posibles valores de i es $i = \{0, 1, 2, ..., n\}$). Los números fraccionarios (reales) sí tienen potencias negativas de la base en el sumatorio.

   Con este tipo de representación, el problema es que leer y escribir largas cadenas de 1's y 0's es engorroso y una tarea propensa a cometer errores. Como solución, se suele utilizar la base octal ($b = 8$) y la base hexadecimal ($b = 16$). Como las dos bases son potencias de dos, las conversiones son triviales.

Dentro de los números enteros tendremos que distinguir a su vez entre los números positivos y negativos.

- Si sólo representamos enteros positivos, con n bits representaremos desde el 0 al $2^n - 1$.
- Para codificar los números enteros negativos, hemos de tener en cuenta algunas consideraciones:
  a) el intervalo de números positivos debería ser igual al intervalo de números negativos para un n dado
  b) sea fácil detectar el signo por un simple test HW
  c) sea también fácil detectar el 0
  d) el código resultante de la codificación ha de dar lugar a una implementación sencilla a la hora de realizar las operaciones aritméticas básicas.

  Para dar respuesta a estas consideraciones, aparecen tres sistemas de numeración posicionales que permiten la representación de números negativos:

a) **Signo/Magnitud**
   - Llamaremos bit de signo al bit más significativo (el primer bit por la izquierda). En función de su valor determinamos el signo del número:
     $\Rightarrow$ 0 $\rightarrow$ número positivo
     $\Rightarrow$ 1 $\rightarrow$ número negativo
   - para n bits, el intervalo de representación va de $-(2^{n-1} - 1)$ a $2^{n-1} - 1$
   - como ventajas, cumple los requisitos (a) y (b).
   - como gran inconveniente, da lugar a una representación dual del 0, con lo cual se desperdicia un número y complica la detección del 0 (requisito (c)). Además no se verifica el requisito (d), porque la suma no es tan evidente como veremos en el tema 5. Podemos adelantar que en el caso de la suma de números de distinto signo, será necesaria la implementación HW de un restador y de un comparador de magnitudes para determinar el signo del resultado.

   Como conclusión, esta representación ampliamente utilizada en los computadores de la 3.ª Generación, en la actualidad no se utiliza. Ahora se emplea únicamente para la codificación de números en punto flotante.

b) **Complemento a 1: (C1)**

- Es un caso particular del complemento a la base menos 1, en el que la base es 2. Representaremos un número de n bits en complemento a uno siguiendo la siguiente fórmula:

$$C1(x) = \begin{cases} x & \text{Si } x \geq 0, \\ (2^n - 1) - |x| & \text{Si } x \leq 0. \end{cases} \tag{2.2}$$

  operación que resulta equivalente a complementar los bits de x cuando x es negativo.
- como ventajas, cumple el requisito (a), puesto que la cantidad de números positivos es igual a la de números negativos. Es decir, el rango va desde el $-(2^{n-1} - 1)$ al $2^{n-1} - 1$. Además, cumple el requisito (b), porque es fácil detectar el signo y cumple el requisito (d), porque facilita las operaciones (restar no es más que complementar el sustraendo y sumar, es decir, el algoritmo y el HW para sumar y restar es común)
- como inconveniente, da lugar a una representación dual del 0 (tanto el 00000 como el 11111), por lo que no cumple el requisito (c).

c) **Complemento a 2: (C2)**

- Es un caso particular del complemento a la base, en el que la base es 2. Representaremos un número de n bits en complemento a dos siguiendo la siguiente fórmula:

$$C2(x) = \begin{cases} x & \text{Si } x \geq 0, \\ 2^n - |x| & \text{Si } x < 0. \end{cases} \tag{2.3}$$

  operación que resulta equivalente a complementar los bits de x y sumar 1 ($C2(x) = C1(x) + 1$) si $x < 0$.
- como ventajas, cumple el requisito (a), puesto que la cantidad de números positivos es prácticamente igual a la de números negativos. Es decir, el rango va desde el -2^{n-1} al $2^{n-1} - 1$. Además, cumple el requisito (b), porque es fácil detectar el signo y cumple el requisito (c), porque no tiene representación dual del 0, así que es fácil detectarlo. Cumple así mismo el requisito (d) como veremos en el tema de algoritmos aritméticos.
- como pequeño inconveniente respecto al C1, se puede argumentar que complementar a 2 es más costoso que complementar a 1 ya que hay que realizar una suma adicional. Sin embargo, el

HW que implementa el C2 se puede construir sin ningún sumador y por otra parte, normalmente no es necesario construir el HW para el C2 explícitamente. Por ejemplo, para realizar la resta, debemos sumar el C2 del sustraendo, pero en la implementación sumamos el C1 del sustraendo poniendo además a 1 el carry de entrada del sumador, con lo que la operación es totalmente equivalente. Es decir:

$$A - B \equiv A + C2(B) \equiv A + C1(B) + (Carry_{in} = 1)$$

Como conclusión, podemos decir que es la representación más utilizada a la hora de representar números enteros negativos.

2. No Posicionales[2]

   a) **BCD (decimal codificado en binario natural)**
   Cada dígito decimal se codifica por 4 bits binarios (16 valores posibles). Ello da lugar a $V_{16/10} = 2.9 \times 10^{10}$ códigos BCD's posibles. El código BCD más utilizado es el BCD natural, en el que la codificación de cada dígito es equivalente a la codificación binaria. Podríamos decir que cada dígito se expresa mediante un nibble en el que se utilizan los pesos 8,4,2,1; pero además la posición del nibble está pesada según potencias de 10.

   - La ventaja de este tipo de representación, es que facilita al hombre la comprensión de los números representados. Además, la conversión de números decimales a BCD es más rápida que la conversión a binario.
   - Como gran inconveniente, la eficiencia de almacenamiento decrece en 10/16 veces respecto de la eficiencia de almacenamiento en binario. Ello es debido a que de los posibles 16 números que podemos representar en binario con 4 bits, en BCD sólo representamos 10 (del 0 al 9). Además, con este tipo de codificación se ocupa más memoria, y las operaciones son más lentas.

   Como conclusión, este tipo de codificación es beneficiosa cuando las operaciones de entrada/salida dominan a las de computación (aunque no suele ser así).

   b) **EXCESO A 3**

[2] En un sentido estricto los códigos comentados bajo este epígrafe no son posicionales ya que el valor del número no se puede calcular directamente a partir de la ecuación 2.1

Es otro BCD construido a partir del BCD natural al que se le suma 3 a cada dígito. Como ventaja, es más cómodo hacer operaciones aritméticas, ya que es autocomplementario, puesto que la representación negativa es igual al C9 del número decimal que representa.

c) **AIKEN**
Es otro código BCD en el que los pesos dentro de cada nibble son 1, 2, 4, 2 en vez de 1, 2, 4, 8. Es autocomplementario.

d) **GRAY**
No es un código BCD. También se conoce con el nombre de binario reflejado. Los códigos de dos enteros consecutivos difieren en un sólo bit. Además es cíclico.

2.2.3. Reales

1. Punto Fijo
   Una posibilidad, a la hora de intentar representar números con decimales, puede consistir en colocar un punto en algún lugar de la cadena de unos y ceros que va a representar nuestro número real. Una vez colocado el punto en una determinada posición ya no se puede modificar, ya que esa decisión se toma normalmente durante el diseño de la máquina. A la derecha del punto las potencias de la base son negativas.
   - Como ventaja, en este sistema de representación los requerimientos HW son relativamente simples.
   - Como inconveniente principal, tanto el rango de valores como la precisión quedan limitados, y el programador o analista numérico que quiera resolver un problema, tendrá que escalar los números de entrada y los resultados intermedios para no salirse del intervalo o rango de valores representables.

   Los números enteros se pueden ver como un caso particular de la notación en punto fijo, es decir, como números en punto fijo, pero colocando el punto detrás del bit menos significativo.
2. Punto Flotante
   Como hemos dicho, en la aritmética de punto fijo podemos perder precisión y dígitos significativos con facilidad. Para solucionarlo se propone la notación científica o de punto flotante. Consiste en representar el número mediante una mantisa, una base y un exponente. Puesto que la base es conocida y constante, basta con almacenar la mantisa y el exponente.

En esta representación, el punto decimal "flota", es decir, cambia de posición dependiendo del exponente. Como en el computador la representación ha de ser única, se requiere una representación normalizada. En principio podemos considerar la representación normalizada como aquella que sitúa el dígito más significativo distinto de cero a la derecha del punto.

Antes de entrar en detalle sobre distintos tipos de representación en punto flotante, y para dar una perspectiva histórica sobre este aspecto, veamos que pensaba von Neumann acerca de la representación en punto flotante: von Neumann decía que los propósitos del punto flotante parecían ser, por un lado, retener tantos dígitos significativos como fuera posible y, por otro, obviar los factores de escala. Pero según su punto de vista, opinaba que estas ventajas eran ilusorias por las siguientes razones:

- el tiempo que lleva programar el escalado es menor que el tiempo para programar la operación completa (se puede decir que despreciable en aquella época).
- además, en vez de reservar un campo para representar el exponente, sería conveniente utilizar los bits de ese campo para ampliar la mantisa, consiguiendo así más precisión.
- los números flotantes no sólo ocuparían más memoria, sino que además necesitan una lógica (HW) más costosa y compleja.

La experiencia y la historia han demostrado que von Neumann se equivocó. Hay que tener en cuenta que para los antiguos diseñadores de computadores era preferible ahorrar memoria y tiempo de ejecución a ahorrar tiempo de programación.

Existe una gran cantidad de formatos en punto flotante, en cada uno de los cuales se especifican el número de bits para la mantisa, el número de bits para el exponente, la base, el convenio de normalización, etc. Nosotros veremos dos posibilidades:

a) Formato IEEE 754
b) Formato del IBM/360

A) FORMATO PUNTO FLOTANTE IEEE 754

Hasta 1985 los formatos en punto flotante cambiaban de una familia de computadoras a la siguiente. Ello daba lugar a gran cantidad de problemas para los analistas numéricos que no podían portar sus programas de unas máquinas a otras, así como para los arquitectos de computadores que no sabían que formato implementar y mejorar. Para evitar estos problemas se adopta un formato estándar para números en punto flotante de 32 y 64 bits, el denominado IE^3 754.

A-1) Formato IE^3 754 simple precisión (32 bits)

La palabra de 32 bits se organiza en los siguientes campos:

- 1 bit para el signo (S)
- 8 bits para la característica (C)
- 23 bits para la mantisa (M)

Puesto que la base es 2, el número representado es:

$$N = (-1)^S \times (1.M) \times 2^{C-127}$$

Donde:

- (S) El bit de signo se interpreta como:
  - ▷ S = 0 $\Rightarrow N \geq 0$
  - ▷ S = 1 $\Rightarrow N \leq 0$
- (M) Es la mantisa normalizada. Inicialmente se aplicó el convenio de normalización de tipo. 0.1xxxx. En una mejora posterior se argumenta: en lugar de forzar un 1 a la derecha del punto decimal, lo que hacemos es colocarlo a la izquierda del punto decimal (es decir, en la forma 1.xxxx). Esta será la representación normalizada. Por otro lado, como sabemos que el número está normalizado y que hay un 1 en esa posición, éste no lo representamos. Es decir, como sabemos que el número debe estar normalizado para representarlo, y que si está normalizado seguro que hay un 1 a la izquierda del punto ¿para qué habríamos de desperdiciar un bit de la mantisa almacenando un 1 que siempre va a estar ahí? Por tanto el 1 a la izquierda del punto no se almacena explícitamente en la representación, sino que queda implícito en el convenio.
- (C) La característica representa el exponente en exceso 127. Es decir, el exponente, E, se calcula como:

$$E = C - 127 \Rightarrow C = E + 127$$

Existen dos motivos para representar el exponente mediante una notación en exceso, en lugar de utilizar una notación en C2 o alguna otra que permita representar exponentes negativos:

- En primer lugar, mediante la representación del exponente en exceso, la comparación de dos exponentes para decidir cual es mayor de los dos se puede realizar mediante un sencillo comparador de enteros positivos. La comparación de exponentes es una operación muy frecuente en aritmética flotante y el HW que la implemente debe ser simple y rápido.
- En segundo lugar, también encontramos una cuestión semántica para utilizar notación en exceso: a medida que el número va haciéndose más pequeño, la característica también va haciéndose más pequeña. Como el 0 es el más pequeño, es el número que ha de tener la característica mínima. Utilizando una notación en exceso, la magnitud de la característica decrece cuando decrece la magnitud del número representado, como vemos en la figura 2.1. Si utilizásemos una notación en C2 (por ejemplo) para la característica, la magnitud de la característica decrece cuando decrece la magnitud del número representado si éste es mayor que uno, pero crece cuando el número es menor que uno.

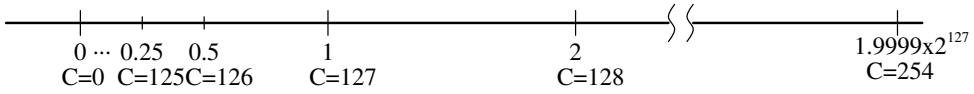


Figura 2.1: Recta real y comportamiento de la característica

De manera que el comportamiento de la característica y del exponente es el siguiente (suponiendo una mantisa 1.0):

$$\begin{array}{lcll} C > 127 & \Rightarrow & E > 0 & \Rightarrow \text{ Número} > 1 \\ C = 127 & \Rightarrow & E = 0 & \Rightarrow \text{ Número} = 1 \\ C < 127 & \Rightarrow & E < 0 & \Rightarrow \text{ Número} < 1 \\ C = 0 & \Rightarrow & --- & \Rightarrow \text{ Número} = 0 \end{array}$$

Casos particulares

Para representar C tenemos 8 bits, es decir, podemos representar en principio 256 (0-255) características distintas. O, traducido a exponente, desde -127 a 128. Sin embargo, reservamos C=0 y C=255 para casos especiales:

$$\begin{cases} C = 255 & \begin{cases} M = 0 \begin{cases} S = 0 \Rightarrow +\infty \\ S = 1 \Rightarrow -\infty \end{cases} \\ M \neq 0 \Rightarrow \text{NaN (Not a Number)} \end{cases} \\ C = 0 & \begin{cases} M = 0 \Rightarrow N = 0 \\ M \neq 0 \Rightarrow N = (-1)^S \cdot 0.M \cdot 2^{-126} \end{cases} \end{cases}$$

Los valores NaN (la traducción literal sería "no un número") aparecen cuando el resultado de una operación no es un número real ($\sqrt{-1}$, 0^0, $\ln(-1)$, etc...). Por otro lado, el caso en que $C = 0$ y $M \neq 0$ se utiliza para representar números no normalizados o desnormalizados. Este caso particular aparece para representar números con exponente menor que -126. Luego aquellos números con $C = 0$ y $M \neq 0$ son números con exponente E=-126 y cuya mantisa no está normalizada, es decir, sin 1 oculto, o lo que es lo mismo, la mantisa es exactamente la almacenada. Los números no normalizados se emplean para representar números muy cercanos a 0.

A-2) Formato IE^3 754 doble precisión (64 bits)

El formato IE^3 754 de doble precisión utiliza una palabra de 64 bits organizada en los siguientes campos:

- 1 bit para el signo (S)
- 11 bits para la característica, (C), en exceso 1023
- 52 bits para la mantisa (M)

Luego el número representado sera:

$$N = (-1)^S \times (1.M) \times 2^{C-1023}$$

B) FORMATO IBM/360

Es el formato implementado en el IBM/360. Se basa en una palabra de 32 bits organizada en los siguientes campos:

- 1 bit para el signo (S)
- 7 bits para la característica (C)
- 24 bits para la mantisa (M)

Pero ahora la base es 16, el exponente en exceso 64 y no existen casos especiales. Por tanto, el número representado vendrá dado por la siguiente expresión:

$$N = (-1)^S \times (0.M) \times 16^{C-64}$$

Para que el exponente crezca o decrezca una unidad, los desplazamientos serán de 4 bits. El número estará normalizado cuando el primer dígito **hexadecimal** a la derecha de la coma sea distinto de 0. Ello dará lugar a que desde el punto de vista binario (que será como se almacene finalmente el número), algunos números no estén normalizados, porque puede haber hasta tres 0's a la derecha de la coma. De esta forma se pierde precisión. La ventaja de esta representación, es que el rango representado aumenta.

2.2.4. Datos No Numéricos. Caracteres

El objetivo es codificar 26 letras mayúsculas y minúsculas, 10 dígitos, signos de puntuación y algunos caracteres especiales (símbolos matemáticos, letras griegas..). También es necesario representar caracteres de control tales como CR (Carriage Return), BS (Back Space), Del (Delete), etc, y caracteres de transmisión como ACK (Acknowledge), NAK (No Acknowledge), EOT (End Of Transmision), ETB (End of Transmision of Block), etc.

Para codificar estos caracteres, se emplean:

- 6 bits. Da lugar a 64 caracteres distintos. Son pocos.
- 7 bits. Da lugar a 128 caracteres. Uno de los códigos más estándares de 7 bits es el conocido como ASCII (American Standard Code for Information Interchange). En la tabla 2.1 encontramos los 128 caracteres ASCII con su codificación en octal.
- 8 bits. Da lugar a 256 caracteres. Un código de 8 bits definido por IBM es el EBCDIC. En la tabla 2.2 tenemos el código EBCDIC completo. En esta tabla los bits 0-3 son los más significativos (por ejemplo, la letra D se codifica 1100 0100).

En muchas máquinas se extiende el ASCII a 8 bits, pero sólo son estándares los primeros 128 caracteres.

2.3. REPRESENTACIÓN DE LAS INSTRUCCIONES

Trataremos, en esta sección, qué convenios se utilizan para representar e interpretar las instrucciones en un computador. Es decir, cada computador tiene

0	NULL	Nulo	20	DLE	Escape de transmisión
1	SOH	Comienzo de cabecera	21	DC1	Mando aparato aux. 1
2	STX	Comienzo del texto	22	DC2	Mando aparato aux. 2
3	ETX	Fin del texto	23	DC3	Mando aparato aux. 3
4	EOT	Fin de la comunicación	24	DC4	Mando aparato aux.4
5	ENQ	Pregunta	25	NAK	Acuse recibo negativo
6	ACK	Acuse de recibo	26	SYN	Sincronización
7	BEL	Timbre	27	ETB	Fin bloque transmitido
10	BS	Retroceso	30	CAN	Anulación
11	HT	Tabulación horizontal	31	EM	Fin del soporte
12	LF	Cambio de renglón	32	SUB	Sustitución
13	VT	Tabulación vertical	33	ESC	Escape
14	FF	Present. formulario	34	FS	Separador archivo
15	CR	Retroceso del carro	35	GS	Separador grupo
16	SO	Fuera de código	36	RS	Separador artículo
17	SI	En código	37	US	Separador subartículo

40	Espacio	60	0	100	@	120	P	140		160	p
41	!	61	1	101	A	121	Q	141	a	161	q
42	”	62	2	102	B	122	R	142	b	162	r
43	#	63	3	103	C	123	S	143	c	163	s
44	$	64	4	104	D	124	T	144	d	164	t
45	%	65	5	105	E	125	U	145	e	165	u
46	&	66	6	106	F	126	V	146	f	166	v
47	’	67	7	107	G	127	W	147	g	167	w
50	(	70	8	110	H	130	X	150	h	170	x
51	)	71	9	111	I	131	Y	151	i	171	y
52	*	72	:	112	J	132	Z	152	j	172	z
53	+	73	;	113	K	133	[	153	k	173	{
54	,	74	<	114	L	134	\	154	l	174	|
55	-	75	=	115	M	135	]	155	m	175	}
56	.	76	>	116	N	136	’	156	n	176	~
57	/	77	?	117	O	137	_	157	o	177	Del

Tabla 2.1: Código de caracteres ASCII en octal

un conjunto de instrucciones, y éste debe ser codificado de alguna manera mediante cadenas de unos y ceros. Pero además, como vimos en el capítulo 1, las instrucciones deben especificar, no sólo el código de operación, sino también los operandos.

2.3.1. Formatos de instrucciones. Introducción

El formato de las instrucciones es el modo en que organizamos una cadena de 1's y 0's para que nos dé información tanto del código de operación como de los operandos. Dentro de esa cadena de 1's y 0's debe existir, por tanto, un campo que llamaremos C.O. (código de operación) con el suficiente número de bits para poder codificar todos los posibles códigos de operación. Además, deben

Lo/Hi	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-						{	}	\	0
1	SOH	DC1	SOS						a	j	~		A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	DC3							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	$\frac{EOB}{ETB}$	UC					f	o	w		F	O	W	6
7	DEL	IL	$\frac{PRE}{ESC}$	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9	RLF	EM						\	i	r	z		I	R	Z	9
A	SMM	CC	SM		ç	!	|	:							
B	VT				.	$	’	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	’								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	|	¬	?	”								

Tabla 2.2: Código EBCDIC completo, de 8 bits. Por ejemplo, la letra J se representa por D1h (primero el código de la columna seguido del de la fila)

existir campos adicionales para especificar el número de operandos necesarios. Entenderemos por operandos tanto los datos a la entrada (argumentos de la operación) como a la salida (resultados de la operación).

En función de los modos de direccionamiento (como veremos en el siguiente epígrafe) en el campo “operando” podemos almacenar directamente el dato con el que vamos a operar; o la dirección de memoria donde se encuentra el dato; o el registro del procesador donde está el dato, etc.

Uno de los problemas que podemos encontrar al diseñar el formato de las instrucciones, aparece si estas instrucciones operan sobre una gran cantidad de operandos, digamos n. En ese caso, el formato de la instrucción tendrá n campos “operando” y, por tanto, necesitaremos bastantes bits para codificar las instrucciones. Si queremos que las instrucciones no ocupen tanta memoria podemos hacer los siguiente: vamos a especificar de forma explícita m operandos (donde $m < n$) y los $n - m$ operandos restantes que espera la instrucción, deben ser colocados en ciertos registros predeterminados. De esta forma, el número de campos “operando” del formato de la instrucción se ve reducido de n a m.

2.3.2. Modos de direccionamiento

Existen diversas formas de especificar los operandos en una instrucción. A estas distintas formas se les denomina modos de direccionamiento. Los tres modos de direccionamiento básicos son el inmediato, directo e indirecto. Vamos a comentar estos tres modos de direccionamiento para la misma instrucción: por ejemplo, la instrucción "LOAD operando" cuya misión es cargar el operando en un registro que llamaremos "acumulador" o AC.

1. **Direccionamiento inmediato**

   Si el operando es una constante, podemos especificarlo directamente en la instrucción.

   Ej.: LOAD $x \Rightarrow$ en el acumulador se cargará el número x.

2. **Direccionamiento directo**

   No siempre trabajamos con constantes. En los programas se suele trabajar con variables, que son aquellos datos cuyo valor varía en el transcurso de la ejecución del programa. La implementación de una variable en el nivel máquina se traduce en reservar una zona de almacenamiento fija donde se guarda el dato (por ejemplo, una posición de memoria o un registro -que es más rápido-).

   Cuando hacemos referencia a una variable, utilizaremos el direccionamiento directo. Es decir, el campo "operando" nos proporcionará la dirección de memoria (o el registro en su caso) donde se encuentra el dato con el que queremos operar.

   Ej.: LOAD $[x] \Rightarrow$ trata x como la dirección de memoria donde está el valor que se quiere cargar en el acumulador.

   Diremos que el direccionamiento es **directo a memoria** si especificamos la dirección de memoria donde está el operando. Diremos que el direccionamiento es **directo a registro** si especificamos el registro en el que se encuentra el operando.

   El direccionamiento directo a registro es evidentemente más rápido que el direccionamiento directo a memoria. Pero, además, el número de bits del campo "operando" será menor con el direccionamiento a registro, ya que hay menos registros que posiciones de memoria y, por tanto, los registros se direccionan con pocos bits.

3. **Direccionamiento indirecto**

   A veces no trabajamos con variables, sino con punteros a variables. Un puntero no es más que otra variable que contiene la dirección de memoria donde se encuentra almacenada una variable. El direccionamiento indirecto nos permite trabajar con punteros.

Ej.: LOAD $[[x]] \Rightarrow$ trata x como la dirección de memoria donde está la dirección del operando[3].

Diremos que el direccionamiento es **indirecto a memoria** si especificamos la dirección de memoria donde está la dirección del operando. Diremos que el direccionamiento es **indirecto a registro** si especificamos el registro donde está la dirección del operando.

El modo de direccionamiento puede estar indicado de forma implícita en el código de operación o de forma explícita mediante un campo adicional. Evidentemente, también se debe especificar de alguna forma (mediante un campo adicional en el formato de instrucción o de forma implícita en el C.O.) si el direccionamiento es a registro o es a memoria.

Como es fácil suponer, el principal inconveniente del direccionamiento indirecto es su lentitud, ya que necesita un primer acceso (a memoria o a registro) para conseguir la dirección del operando, y un segundo acceso a memoria para conseguir el operando en sí.

2.3.3. Modificaciones sobre los modos de direccionamiento

Llamaremos a partir de ahora **dirección efectiva** a la dirección en la que se encuentra el operando. En general, podemos decir que los distintos tipos de direccionamiento se distinguen unos de otros en el modo en que se calcula la dirección efectiva (es decir, en el modo en el que se determina donde está el operando).

Sobre los tres modos de direccionamiento básicos vistos anteriormente, podemos definir algunas variaciones y combinaciones como las que veremos ahora:

1. El direccionamiento directo a memoria puede ser:
   - Direccionamiento **absoluto**: el campo operando proporciona la dirección completa del operando en memoria. Para especificar la dirección completa necesitamos que el campo "operando" tenga un número de bits igual al del bus de direcciones. Es decir, el tamaño de las instrucciones que utilizan direccionamiento absoluto es grande.
   - Direccionamiento **relativo**: el campo operando contiene un desplazamiento respecto una posición conocida (dirección base) que almacenamos en un registro determinado (registro base, registro

[3]Modificamos la sintaxis de la instrucción (de LOAD a LOAD[[...]]) para indicar que utiliza direccionamiento indirecto. Es decir, el código de operación de la instrucción LOAD[[...]] debe ser distinto al de la instrucción LOAD, y para hacérselo notar al ensamblador utilizaremos esta sintaxis.

de dirección, contador de programa, etc). El direccionamiento relativo al Contador de Programa (PC), se emplea cuando los datos están cerca de la instrucción que se está ejecutando. En este tipo de direccionamiento la dirección efectiva se calcula sumando el desplazamiento a la dirección base.

Ventajas del direccionamiento relativo:

- Se reduce la longitud de la instrucción.
- Podemos mover o relocalizar un bloque de memoria en otra zona. Por ejemplo, un programa donde sólo se utilice direccionamiento relativo al PC funcionará de la misma forma independientemente de cual sea la dirección inicial en la que esté cargado el programa. Otro ejemplo lo podemos encontrar en el manejo de vectores: si queremos realizar una misma secuencia de accesos sobre dos vectores almacenados a partir de dos posiciones distintas de memoria, podemos escribir un código que realice esa secuencia de accesos relativa al comienzo de un vector y, posteriormente, repetir la ejecución del mismo código modificando únicamente la dirección de comienzo del vector. En este caso, llamaremos **dirección base** a la dirección de comienzo del vector, y **registro base** al registro que almacena la dirección base. A este tipo de direccionamiento se le conoce con el nombre de **direccionamiento relativo a base**.
- Facilita el acceso secuencial a zonas de almacenamiento consecutivo (vectores, matrices y arrays en general). Este tipo de acceso se consigue utilizando una constante que apunta al comienzo de un vector, a la que se suma el contenido de un registro (que llamaremos **registro índice**). El registro índice especifica qué "índice" del vector es el que va a ser accedido. A este tipo de direccionamiento se le conoce con el nombre de **direccionamiento indexado**. Un caso particular del direccionamiento indexado es el direccionamiento **autoindexado**. En este modo, el registro índice se incrementa o decrementa (dependiendo de como lo especifique el programador) automáticamente después de cada acceso. Los dos modos básicos de direccionamiento autoindexado son:
  - **con postincremento**: el registro índice se autoincrementa después del acceso al operando.
  - **con predecremento**: el registro índice se autodecrementa antes del acceso al operando.

Inconvenientes del direccionamiento relativo:

- Aumento de los circuitos lógicos para calcular la dirección efectiva, lo que supone mayor coste.
- Aumento del tiempo de cálculo de la dirección efectiva.

2. Direccionamiento de Pila

   En ciertas ocasiones resulta útil emplear la memoria principal como si fuese una "pila". Una **pila** es una estructura LIFO: los datos están accesibles sólo en el tope de la pila. La pila se direcciona implícitamente mediante el SP (Stack Pointer). El SP siempre apunta a la ultima posición llena de la pila. Las instrucciones asociadas a una estructura pila utilizan el direccionamiento de pila. Estas instrucciones son:

   - PUSH → mete un dato en la pila
   - POP → saca un dato de la pila

2.3.4. Número de direcciones

El problema que abordaremos ahora es el de determinar cuántas direcciones u operandos proporcionamos a las instrucciones. Las posibilidades que podemos encontrar en máquinas reales son:

- 3 direcciones: especificamos de forma explícita dos operandos fuentes y un destino.

  Ej: ADD x,y,z ⇒ El resultado de y+z se guarda en x (p.e. en varios procesadores RISC como el R10000, el Alpha, etc.).
- 2 direcciones: perdemos un operando ya que especificamos de forma explícita los dos operandos fuentes, pero el destino está implícito en uno de los dos anteriores.

  Ej: ADD x,y ⇒ El resultado de x+y se guarda en x (p.e. en los procesadores de Intel) o en y (p.e. en los procesadores de Motorola).
- 1 dirección: Un operando fuente se proporciona de forma explícita, pero el otro operando, así como el resultado, quedan implícitos.

  Ej: ADD x ⇒ El resultado de x+AC se guarda en AC (suponiendo que en la máquina en que se ejecuta esta instrucción, el convenio es que el registro AC contenga los operandos implícitos, como en el NEC 78000).
- 0 direcciones: Todos los operandos son implícitos, por ejemplo, cuando se trabaja con una pila.

  Ej: ADD ⇒ El resultado de sumar los dos operandos que están en la cima de la pila, se coloca en la cima de la pila.

Tener instrucciones con un número elevado de direcciones presenta ventajas e inconvenientes:

- Ventajas:
  1. Los programas son más cortos lo que supone un menor gasto de memoria
  2. Las instrucciones son más potentes y los tiempos de ejecución menores.
- Inconvenientes:
  1. Las instrucciones resultantes son más anchas, lo que supone gasto de memoria.
  2. Es necesaria una lógica adicional de decodificación.

Normalmente los procesadores potentes, y en particular los que siguen una arquitectura RISC, utilizan un formato de instrucciones de tres direcciones. En general, no suele ser interesante tener un número mayor de direcciones. En procesadores más simples se toma como compromiso utilizar un formato de dos direcciones.

Por otro lado, el número de direcciones puede variar de unas instrucciones a otras, en función de cuantos operandos necesite. Por ejemplo, a la instrucción NEG, que calcula el complemento a dos de un operando, no tiene sentido darle un formato de tres direcciones.

2.3.5. Códigos de operación

El código de operación es el campo de la instrucción que especifica la operación a realizar. Si el modo de direccionamiento no está en un campo aparte, también lo especificaría el código de operación. Es evidente que si el campo reservado al C.O. tiene k bits, se pueden representar 2^k C.O. distintos.

Una de las técnicas más empleadas para reducir el tamaño de los programas consiste en asignar códigos de operación más cortos a las instrucciones más frecuentes. Es una técnica normal de codificación (códigos Huffmann, código Morse).

El código de operación no tiene por qué tener una longitud fija. Es lo que se denomina C.O. con extensión. Dependiendo de las instrucciones y de los operandos que necesiten, los C.O. pueden ser más largos o más cortos.

2.3.6. Tipos de instrucciones

No olvidemos que aunque programemos en alto nivel, las máquinas no entienden directamente estos lenguajes, sino que ejecutan instrucciones máquina. Es necesario un compilador para que traduzca cada sentencia de alto nivel en instrucciones de código máquina. Elegir un conjunto de instrucciones apropia-

do facilitará la tarea de la compilación y dará lugar a códigos ejecutables de mayores prestaciones.

Los requerimientos que debe cumplir un conjunto de instrucciones máquina son:

1. Debe ser completo. Es decir, con el conjunto de instrucciones máquina se ha de poder programar cualquier función computable utilizando una cantidad de memoria razonable.
2. Debe ser eficiente. Las funciones más requeridas se deben ejecutar mediante pocas instrucciones rápidamente.
3. Debe ser regular. Deben existir los C.O. y modos de direccionamiento esperados (por ejemplo, si existe desplazamiento a la derecha, debe existir también el desplazamiento a la izquierda).
4. Las instrucciones han de ser compatibles con las de procesadores de la misma familia. De esta manera se reducen costes en el desarrollo SW.

Normalmente, el conjunto de instrucciones es distinto para procesadores de casas o series distintas. Sin embargo, podemos decir que todos los procesadores tienen instrucciones de los cinco tipos siguientes:

1. Instrucciones de transferencia (movimiento) de datos.
   Ej.: LOAD, MOVE, PUSH, etc.
2. Instrucciones aritméticas: operaciones sobre datos numéricos.
   Ej.: ADD, MUL, etc.
3. Instrucciones lógicas: operaciones booleanas y no numéricas.
   Ej.: NOT, AND, OR, etc.
4. Instrucciones de control de programa: saltos, saltos condicionales, llamadas a subrutinas.
5. Instrucciones de E/S: transferencia de datos entre el mundo exterior y el procesador (un caso particular de las instrucciones de transferencia de datos).

SINOPSIS

En este tema hemos analizado los convenios a los que han llegado los arquitectos de computadores para representar la información mediante 1's y 0's. Información que va desde datos numéricos, números negativos, números reales por medio de un exponente y una mantisa, caracteres, hasta instrucciones, operandos, etc. Así mismo hemos abordado el problema del acceso a los datos, mediante el estudio de los modos de direccionamiento. Y todo sólo mediante 1's y 0's.

RELACIÓN DE PROBLEMAS

Representación de los datos

1. Convierte los siguientes números decimales a binario: 1984, 4000, 8192.
2. ¿Qué número es $1001101001_{(2}$ en decimal? ¿En octal? ¿En hexadecimal?
3. ¿Cuáles de éstos son números hexadecimales válidos?: CAE, ABAD, CEDE, DECADA, ACCEDE, GAFE, EDAD. Pasar los números válidos a binario.
4. Expresa el número decimal 100 en todas las bases, de 2 a 9.
5. ¿Cuántos enteros positivos se pueden representar con k dígitos usando números en base r?
6. El siguiente conjunto de 16 bits:

   1001 1000 0101 0100

   puede representar un número que depende del convenio utilizado. Dar su significación decimal en el caso de que el convenio sea:
   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2
   d) BCD natural (8,4,2,1)
   e) BCD exceso a 3
   f) BCD Aiken (2,4,2,1)
7. Para las siguientes representaciones de n bits, dar el rango, precisión, representaciones del 0 y comparar el número de números positivos con el de negativos.
   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2
8. Expresar los números decimales -63, 91 y -23 en signo/magnitud, en complemento a uno y en complemento a dos utilizando una palabra de 8 bits.
9. ¿Cuántos bits son necesarios para representar todos los números entre -1 y +1 con un error no mayor de $0.0001_{(10}$ en complemento a dos y con notación en punto fijo?
10. Representar los siguientes números como flotantes IEEE-754 en simple precisión.
    a) 10
    b) 10.5

c) 0.1
d) 0.5
e) -23.15625

11. ¿Qué inconveniente encontramos en la representación de números reales en punto fijo?
Dados estos dos números en el formato IEEE 754: C0E80000 y 00080000; decir que valores representan en decimal.
12. ¿Cuáles son los requisitos deseables de un sistema de numeración de enteros con signo? ¿Qué sistema cumple mejor esos requisitos? Expresar el número 9.75 en su representación IEEE 754 de 32 bits (convierte a hexadecimal la palabra IEEE de 32 bits).
13. En una PDP-11 los números en punto flotante de precisión sencilla tienen un bit de signo, un exponente en exceso a 128 y una mantisa de 24 bits. Se exponencia a la base 2. El punto decimal está en el extremo izquierdo de la fracción. Debido a que los números normalizados siempre tienen el bit de la izquierda a 1, éste no se guarda; solo se guardan los 23 bits restantes. Expresa el número 7/64 en este formato.
14. Expresa el número 7/64 en el formato IBM/360, y en el formato IEEE-754.
15. Expresa el número en punto flotante de 32 bits $3FE00000_{(16}$ como número decimal si está representado en los siguientes formatos de 32 bits:

a) IEEE-754
b) IBM/360
c) PDP-11

16. Los siguiente números en punto flotante constan de un bit de signo, un exponente en exceso a 64 y una mantisa de 16 bits. Normalízalos suponiendo que la exponenciación es a la base 2 y que el formato NO es del tipo 1.XXXX... con el "1" implícito.
    - 0 1000000 0001010100000001
    - 0 0111111 0000001111111111
    - 0 1000011 1000000000000000
17. ¿Cuál son los números positivos más pequeño y más grande representables para los siguientes convenios?:

a) IEEE-754
b) IBM/360
c) PDP-11

Representación de las instrucciones

1. Diseña un código de operación con extensión que permita lo siguiente y se pueda codificar en una instrucción de 36 bits:
   a) 7 instrucciones con dos direcciones de 15 bits y un número de registro de 3 bits,
   b) 500 instrucciones con una dirección de 15 bits y un número de registro de 3 bits,
   c) 50 instrucciones sin direcciones ni registros.
2. ¿Es posible diseñar un código de operación con extensión que permita codificar lo siguiente en una instrucción de 12 bits? Un registro se direcciona con 3 bits.
   a) 4 instrucciones con tres registros,
   b) 255 instrucciones con un registro,
   c) 16 instrucciones con cero registros
3. Cierta máquina tiene instrucciones de 16 bits y direcciones de 6. Algunas instrucciones tienen una dirección y otras dos. Si hay n instrucciones de dos direcciones, ¿cuál es el número máximo de instrucciones de una dirección?
4. Queremos diseñar el formato de instrucciones de un procesador con 16 registros internos, y que puede direccionar 1 Kbyte de memoria. Dentro del conjunto de instrucciones encontramos las siguientes:
   a) 15 instrucciones del tipo:

Código de Op.	Desp, Dirección, Dirección

   b) 63 instrucciones del tipo:

Código de Op.	Desp, Dirección, Registro

   c) 60 instrucciones del tipo:

Código de Op.	Dirección, Registro

   d) 15 instrucciones del tipo:

Código de Op.	Registro, Registro, Registro

   e) 16 instrucciones del tipo:

Código de Op.

   Donde el campo desplazamiento (*Desp*) nos permite implementar el direccionamiento relativo, desplazándonos como máximo 63 bytes respecto de la dirección base apuntada por la primera *Dirección*. El desplazamiento siempre será positivo.

   Se pide:

a) Decir de cuántas direcciones es cada tipo de instrucciones (0, 1, 2 ó 3).
b) Diseñar el formato de cada tipo de instrucciones mediante códigos de operación con extensión.

5. Dados los contenidos de las celdas de memoria que siguen y una máquina de una dirección con un acumulador, ¿qué valores cargan en el acumulador las instrucciones siguientes?
   - la palabra 20 contiene 40
   - la palabra 30 contiene 50
   - la palabra 40 contiene 60
   - la palabra 50 contiene 70

   a) LOAD 20
   b) LOAD [20]
   c) LOAD [[20]]
   d) LOAD 30
   e) LOAD [30]
   f) LOAD [[30]]

6. Compara las máquinas de 0, 1, 2, y 3 direcciones escribiendo programas para calcular

$$X = (A + B \times C)/(D - E \times F)$$

   para cada una de las cuatro máquinas. Para que no se pierdan los valores originales de las variables A, B, C, D, E y F podremos apoyarnos en una variable temporal T. Suponiendo direcciones de 16 bits, códigos de operación de 8 bits y longitudes de instrucción que son múltiplos de 4 bits, ¿Cuántos bits necesita cada computadora para calcular X?

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

3 Procesador Central

OBJETIVOS

- Estudiar a nivel RTL la arquitectura del procesador central
- Introducir los conceptos de subrutina, interrupciones y segmentación o *pipeline*
- Dar unos primeros pasos en el campo del lenguaje ensamblador

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

3.1. INTRODUCCIÓN. ESTRUCTURA BÁSICA DE UN PROCESADOR

Volvamos por un momento a la estructura de la máquina de von Neumann, reorganizando ahora las unidades del computador desde un punto de vista más moderno, como el presentado en la figura 3.1.

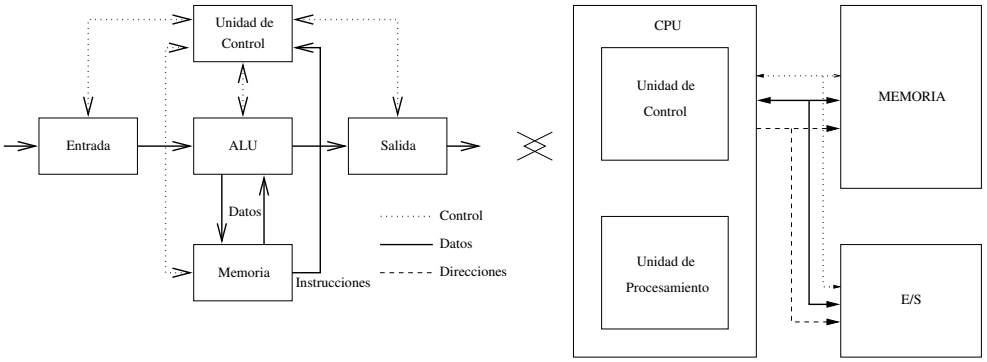


Figura 3.1: Unidades del computador

Como vemos, el computador se compone de los siguientes módulos:

1. **Unidad Central de Proceso (UCP o CPU)**. Compuesta por una unidad de control y otra de procesamiento (ALU más registros). Normalmente se construye sobre una pastilla de silicio que llamaremos procesador o microprocesador (μP).
2. **Memoria**. Subsistema en el que residen los programas y los datos. Este subsistema se organiza en varios niveles conformando lo que se conoce con el nombre de jerarquía de memoria: caché, memoria principal y memoria secundaria (disco duro).
3. **Unidad de Entrada/Salida**. Mediante la cual se controla la entrada y salida de información. Gestiona el manejo de periféricos en general: teclado, monitor, impresora, ratón, etc.

Estas tres unidades se encuentran interconectadas mediante buses de comunicación. Estos buses, no son más que líneas que transportan información binaria entre los diversos subsistemas. Los podemos clasificar en función del tipo de información que llevan:

Bus de direcciones. Mediante el cual la CPU selecciona la posición de memoria o la unidad de E/S de la cual va leer información o en la cual va a

escribir información. Evidentemente, si el bus tiene n líneas, tendremos acceso a 2^n posiciones distintas (a repartir entre RAM, ROM y E/S). Este bus es unidireccional.

Bus de datos. Mediante el cual se transfiere información de (hacia) la memoria o E/S hasta (desde) el procesador. Es un bus bidireccional y normalmente determina la palabra de trabajo del procesador (si el bus tiene n líneas decimos que el procesador es de n bits).

Bus de control. Mediante el cual se transfiere información de control entre el procesador, memoria y E/S. Es un bus bidireccional.

Podemos caracterizar un bus por el número de líneas de comunicación en paralelo (bits) que lo componen y por la frecuencia de transmisión de los datos en Hz. Ambos parámetros pueden en este caso fundirse en una sola magnitud, que actúa como métrica de su rendimiento, y que se conoce como *ancho de banda* (BW, del inglés "bandwidth"), indicando la velocidad de comunicación por el bus en bits por segundo (bps.). Por ejemplo, el bus local PCI del Pentium tiene una anchura de 64 líneas y una frecuencia de 66 MHz., lo que proporciona un ancho de banda de 528 Mbytes/s.

En este tema nos centraremos en el procesador central de forma general. Es el componente más complejo del sistema y su misión es sincronizar y coordinar el funcionamiento de los demás módulos del sistema, leyendo, decodificando y ejecutando instrucciones.

3.1.1. Características principales del procesador

1. **Longitud de palabra.**

   Definida por el n.º de bits del bus de datos, bus de direcciones, tamaño de los registros o por el n.º de bits con que la ALU puede operar en paralelo. En los procesadores modernos la longitud de palabra impone un limite máximo al rango de direccionamiento de memoria.

   Normalmente los registros de la máquina tienen la misma anchura, pero no tiene porqué ser así. Por ejemplo, el 68000 de Motorola tiene registros de 32 bits, pero tiene un bus de datos de 16 bits. El Intel 8088 tiene registros de 16 bits y un bus de 8 bits.

   De cualquier modo, el que la longitud de palabra sea 32 bits (por ejemplo), no quiere decir que toda la información en este μP particular ocupe 32 bits. Es decir, podemos codificar caracteres con 8 bits, códigos de operación con 16 bits y números en punto flotante de doble precisión con 64 bits.

   En general nos referiremos con *longitud de palabra del procesador* al

número de bits de los registros accesibles al programador (anchura de la ALU).

2. **Velocidad de proceso.**

   Estará en función de ciertos parámetros de diseño:

   - Frecuencia de reloj (33MHz, 100MHz, 200MHz, etc)
   - Tiempo de ejecución de cada instrucción: CPI (ciclos de reloj por instrucción)
   - Tiempo de acceso a memoria y a E/S. Normalmente estas dos unidades ralentizan en gran medida al sistema completo.

   El tiempo que un procesador invierte en la ejecución de un programa puede obtenerse como el producto de tres factores:

   a) NI: Número de instrucciones máquina en que se transforma el programa.
   b) CPI: Número medio de ciclos de reloj que se necesitan para ejecutar cada instrucción máquina.
   c) T: Tiempo de ciclo de reloj (o su frecuencia f como magnitud inversa).

   Es decir, $T_{CPU} = NI \times CPI \times T = \frac{NI \times CPI}{f}$, donde T_{CPU} resultará en la misma magnitud en que expresamos el período T, por ejemplo, en microsegundos si la frecuencia f viniera expresada en MHz.

   El primero de esos factores depende principalmente del compilador. Si éste está optimizado para un procesador, será capaz de traducir un programa de alto nivel (escrito en C, por ejemplo) en un programa objeto o ejecutable que contenga el menor número de instrucciones máquina posibles.

   El segundo factor, CPI, está más relacionado con el conjunto de instrucciones del procesador. Por ejemplo, un procesador RISC, (cuyo conjunto de instrucciones se compone de pocas instrucciones muy sencillas) puede reducir el CPI incluso a la unidad, pero a costa de incrementar NI. Para ello, una de las técnicas que utiliza es la de *segmentar* ("pipeline") la ejecución de instrucciones dentro del procesador.

   El tercer factor es la frecuencia de reloj del procesador, que está más relacionada con su tecnología hardware y con las técnicas de integración de circuitos. La positiva evolución de éstas ha permitido duplicar la frecuencia del procesador cada 2 ó 3 años en las dos últimas décadas.

3. **Capacidad de proceso.**

   En relación con:

   a) Número de instrucciones del μP, distinguiendo entre los del tipo

CISC (Complex Instruction Set Computer) y RISC (Reduced Instruction Set Computer). Con relación a este parámetro se utiliza la métrica MIPS (millones de instrucciones por segundo) que viene definida como:

$$MIPS = \frac{NI}{T_{CPU}} \cdot 10^{-6}$$

La principal ventaja de los MIPS es que son fáciles de comprender. Sin embargo, son dependientes del repertorio de instrucciones, pues a mayor complejidad de las instrucciones máquina de un procesador, menor número de MIPS producirá un programa concreto, sin que eso indique que el programa se está ejecutando más lentamente.

b) Operaciones que puede realizar la ALU (aritmética entera, BCD, formato en punto flotante de simple o doble precisión, división y multiplicación por HW o por SW, etc...). Los MFLOPS o millones de operaciones en punto flotante por segundo (del inglés *Million of FLoating-point Operations Per Second*), caracteriza la velocidad de la unidad en punto flotante y por otro lado es una métrica que está basada en operaciones en lugar de instrucciones como los MIPS.

4. **Gestión de memoria.**

Se refiere a las características que posee el μP en relación con la gestión de memoria. Por ejemplo, si incluye HW para el manejo de caché, si lleva un nivel de caché interno, si implementa una unidad de manejo de memoria (MMU) para gestionar la memoria virtual, etc.

Los dos parámetros principales que influyen en las prestaciones de una memoria son su *tamaño* en número de palabras y su *latencia* o tiempo de respuesta expresado normalmente en nanosegundos (ns.). El tamaño aumenta la funcionalidad del sistema al permitir ejecutar aplicaciones más complejas, mientras que la latencia está más ligada a la rapidez de ejecución de los programas. El número de bytes que la memoria sea capaz de proporcionar por unidad de tiempo, es decir, su *ancho de banda* vendrá dado como:

$$BW = \frac{longitud\_de\_palabra\_de\_memoria}{latencia}$$

Capacidad y latencia son parámetros inversamente ligados con respecto al precio, es decir, a mayor tamaño, menor rapidez por el mismo precio. Por ejemplo, las memorias caché actuales tienen un tamaño de

256-512 Kbytes y una latencia de unos pocos ns., la memoria principal o RAM presenta un tamaño de entre 16 y 128 Mbytes y una latencia de decenas de ns., y la memoria secundaria o disco duro llega a varios Gbytes, pero sube la latencia a los milisegundos.

5. **Manejo de interrupciones y capacidad de interfaz.**
   Se refiere a las capacidades del μP para relacionarse con otros dispositivos externos. Por ejemplo, como maneja las interrupciones, si soporta dispositivos de acceso directo a memoria (DMA) para acelerar la transferencia de datos entre periféricos y memoria, etc.

3.2. SUBSISTEMAS DE DATOS Y DE CONTROL

Como hemos repetido en otras ocasiones, el procesador engloba las unidades de control, de datos y los registros. En este apartado daremos una primera introducción a estos tres bloques:

3.2.1. Registros

Los registros son dispositivos digitales que nos permiten almacenar información y acceder a ella en tiempos bastante menores que el que necesitaríamos para acceder a memoria. Podemos clasificar los registros que podemos encontrar en un μP en función de la información que almacenan:

1. **Registros de propósito general (RPG):** En estos registros almacenamos la información que utilizamos más frecuentemente (ya que el acceso a memoria es más lento). Podemos distinguir dos subtipos de RPG's:
   - Registros de datos: Almacenan datos y tienen normalmente un número de bits igual al de la palabra del μP.
   - Registros de direcciones: Normalmente almacenan direcciones, por lo que deben tener un tamaño igual al del bus de direcciones.
2. **Contador de Programa (PC o IP):** Contiene la dirección de memoria de la cual se leerá la siguiente instrucción a ejecutar. Normalmente el contenido del PC se incrementa al ejecutar cada instrucción para que "apunte" a la siguiente instrucción a ejecutar. Si modificamos el contenido del PC cargándolo con la dirección x, estamos realizando un "salto" a la instrucción almacenada en la posición x de memoria.
3. **Registro de Instrucciones (RI):** Almacena el código de operación de la instrucción que estamos ejecutando en un momento dado. Es un

registro "transparente al usuario", es decir, el usuario o el programador no pueden acceder a ese registro y modificar su valor, sino que es un registro que actualiza automáticamente la sección de control del μP.

4. **Acumuladores:** En muchos μP hay uno (o varios) registros acumulador (AC) en el que implícitamente hay un operando y donde se "acumula" el resultado. En caso de no tener registro AC, podemos utilizar registros de propósito general.
5. **Registro índice y registro base:** Pueden ser registros de direcciones que se utilizan para el direccionamiento relativo a base (el registro base contiene la dirección base a cierta posición de memoria) o para el direccionamiento indexado (el registro índice va apuntando a las distintas posiciones de una estructura de datos con almacenamiento consecutivo: arrays, cadenas alfanuméricas, etc).
6. **Puntero de Pila (Stack Pointer -SP-):** Puede ser un registro de dirección que contiene el puntero a la posición de la pila escrita más recientemente (la cima de la pila).
7. **Registros temporales:** Normalmente incluidos dentro del μP para guardar resultados intermedios de algunas operaciones (por ejemplo, la instrucción `XCHG`, que intercambia el contenido de dos registros necesita de un registro temporal para realizar el intercambio). Estos registros no suelen ser accesibles por el usuario (son transparentes al usuario).
8. **Registro de estado o de banderas (Flags Reg. o SR):** Es un registro en el que cada bit o campo de bits tienen información independiente, normalmente relacionada con el resultado de las operaciones realizadas en la ALU. A cada uno de esos bits independientes se le llama bandera (flag) y se activan o desactivan en función de la ejecución de ciertas instrucciones. Estos flags son testeados o chequeados por otras instrucciones para realizar saltos condicionales.

   Los flags más comunes son los siguientes:

   a) Cero (Z): Se pone a 1 si el resultado de una operación es 0.
   b) Carry (C): Se pone a 1 si el resultado de una operación provoca un acarreo de salida.
   c) Signo (S): Se pone a 1 si el resultado de una operación es negativo.
   d) Overflow (O): Se pone a 1 si el resultado de una operación se ha salido del rango de valores representables.
   e) Paridad (P): Se pone a 1 si el resultado de una operación da lugar a una palabra con un número par de 1's (o un número impar de 1's, dependiendo del convenio).

Otros flags no son el resultado de realizar operaciones con datos, sino que son activados o desactivados por el programador para obligar al μP a trabajar en uno de varios modos posibles:

a) **Habilitación/Deshabilitación de interrupciones:** El programador pone a 1 este flag si quiere permitir las interrupciones de otros dispositivos. En caso de no aceptar interrupciones, éstas pueden ser enmascaradas poniendo a 0 este flag.
b) **Traza:** A 1 habilita la ejecución de instrucciones paso a paso (modo de depuración de programas).
c) **Supervisor:** A 1 el procesador trabaja en modo supervisor (modo de alta prioridad en el que se pueden ejecutar instrucciones privilegiadas y no hay limitaciones en el acceso a datos -es el modo en que trabaja el Sistema Operativo-). A 0 el procesador trabaja en el modo normal o modo de usuario.

3.2.2. Unidad de Datos

También llamada Sección de Procesamiento, donde el módulo principal es la Unidad Aritmético-Lógica. Este módulo (ALU) contiene la circuitería necesaria para realizar las operaciones aritméticas y lógicas. Normalmente la ALU sólo realiza operaciones enteras, por lo que también se la conoce con el nombre de "unidad entera" para diferenciarla de la "unidad en punto flotante" o FPU[4].

La unidad entera, también es la encargada de calcular la dirección efectiva del operando (ya que en los direccionamientos relativos es necesario realizar sumas o restas para determinar la dirección del operando).

El número de operaciones que puede realizar la ALU y la velocidad de procesamiento dependen en gran medida de la cantidad de HW que utilicemos para implementar las distintas funciones. Podemos abaratar costes en el desarrollo de la ALU emulando ciertas operaciones mediante SW, de forma que no sea necesario incluir un HW específico para esas operaciones. Sin embargo, el tiempo de ejecución de estas instrucciones emuladas mediante SW es mayor.

Como vemos en la figura 3.2 la ALU se alimenta de los registros o de memoria (a través del bus de datos). La salida de la ALU también puede almacenarse en registros o en memoria. También debe existir una entrada de control que indique a la ALU que operación debe realizar (en el caso de la

[4]La FPU también puede estar incluida dentro del μP o colocarse externamente como coprocesador matemático.

figura, las señales c6, c7 y c8 codifican la operación que debe realizar la ALU). Además, la ALU debe actualizar el registro de Estados (SR) después de cada operación.

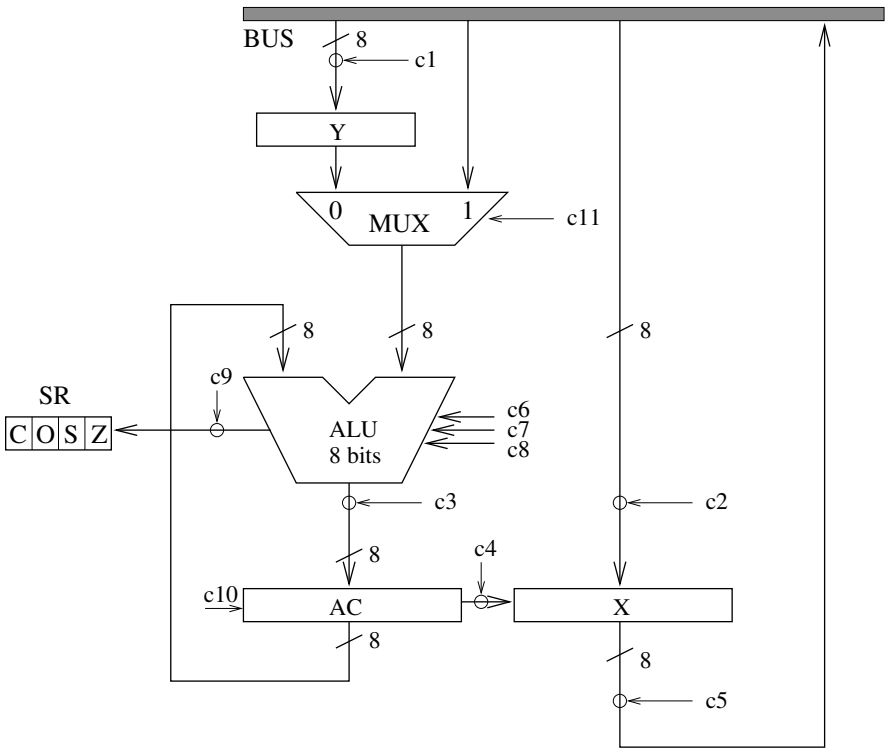


Figura 3.2: Una posible Sección de Procesamiento

Como vemos en la figura 3.2, esta sección de procesamiento tiene varios **"puntos de control"** (de c1 a c11) que deben ser activados adecuadamente para que el secuenciamiento de las operaciones dé lugar a una ejecución correcta de las instrucciones.

El modo particular en el que interconectamos los registros, la ALU, el SR, etc, determina lo que llamaremos el **"flujo de datos"** *(Data Path)*. Este flujo de datos determina cuáles son los posibles caminos que puede tomar la información durante su procesado.

3.2.3. Unidad de Control

También llamada Sección de Control. Es la unidad que genera las señales de control que "atacan" los puntos de control de la sección de procesamiento. Por tanto, esta unidad debe tener la circuitería de control, temporización, y decodificación de instrucciones.

En la figura 3.3 podemos ver un esquema a alto nivel de una sección de control. A partir del contenido del registro de instrucciones, la sección de control debe generar las señales de control para que se ejecute dicha instrucción adecuadamente.

Después del ciclo de búsqueda (fetch), en el registro de instrucción encontraremos la instrucción que vamos a ejecutar. El código de operación de esta instrucción debe ser decodificado durante la fase de decodificación, de forma que la sección de control interprete el objetivo de la instrucción en curso. Una vez decodificada la instrucción, debemos ejecutarla generando de forma secuencial las señales de control que activan las distintas funciones de la sección de procesamiento.

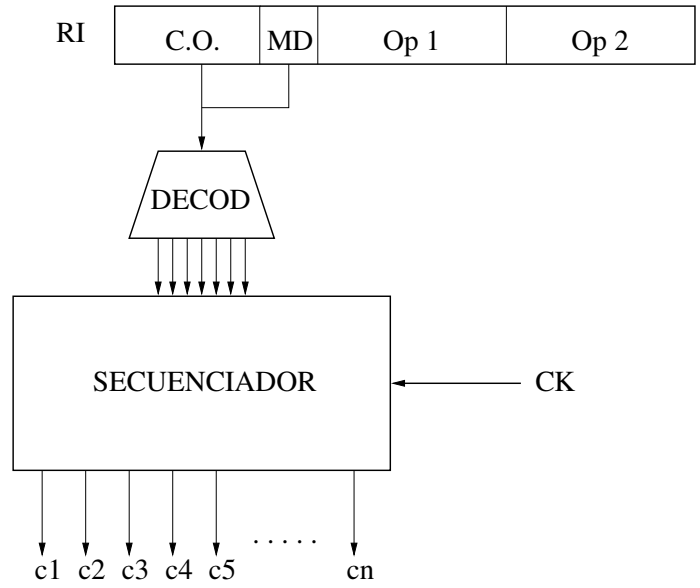


Figura 3.3: Esquema a alto nivel de una sección de control

Para conseguir este objetivo, la sección de control contiene un decodificador y un secuenciador (una máquina de estados secuencial alimentada por una

señal de reloj). El decodificador decodifica el código de operación en curso y genera un conjunto de señales que inicializan el secuenciador colocándolo en un estado inicial. Este estado inicial será distinto para las distintas instrucciones a ejecutar. A partir de este estado inicial el secuenciador evolucionará a través de ciertos estados, cada uno de los cuales generará ciertas señales de salida (señales de control) que activarán los distintos puntos de control de la sección de procesamiento.

A cada conjunto de señales de control para un estado dado se le llama **microinstrucción**, y por tanto la ejecución de cada instrucción máquina (**macroinstrucción**) se traduce en un conjunto de micro-instrucciones. El conjunto de microinstrucciones que ejecutan una determinada macroinstrucción se conoce con el nombre de **microsubrutina**.

Existen básicamente dos técnicas para construir el secuenciador:

1. **Técnica cableada:** El secuenciador se construye mediante una máquina de estados compuesta de puertas, contadores, registros, decodificadores, etc. La interconexión de estos elementos es un poco heurística, poco sistemática y da lugar a secciones de control complicadas, pero bastante rápidas. Es la técnica de control utilizada en los procesadores RISC.
2. **Técnica microprogramada:** El secuenciador se construye mediante un contador de microprograma que apunta a las distintas microinstrucciones almacenadas en una memoria (llamada micro-memoria). Esta técnica es más sistemática y pedagógica, pero da lugar a secciones de control más lentas. Es la técnica utilizada en los procesadores CISC.

3.3. CICLO MÁQUINA Y ESTADOS DE UN PROCESADOR

Como hemos comentado en alguna otra ocasión, la ejecución automática de un programa por un computador se lleva a cabo mediante el procesado de las instrucciones contenidas en el programa. A su vez, la ejecución de estas instrucciones se efectúa por el uso repetitivo de ciertas operaciones del sistema, a saber:

Búsqueda de instrucción. El procesador "busca" en la posición de memoria apuntada por el registro PC una instrucción que carga en el registro RI.

Decodificación. El C.O. de la instrucción se analiza para determinar que función debe realizar la instrucción. En términos más realistas, podemos decir que el C.O. decodificado proporciona un estado inicial a la unidad

de control, de forma que ésta genere la información de control necesaria para ejecutar esta instrucción en particular. También en esta fase se incrementa el PC para que apunte a la siguiente instrucción a ejecutar.

Cálculo de la dirección efectiva. En caso de que la instrucción necesite operandos almacenados en memoria, se determina la dirección en la que se encuentran dichos operandos: la dirección efectiva o EA (*efective address*).

Búsqueda de operandos. Acceso a memoria para leer los operandos de la instrucción.

Ejecución. Realización de la función especificada por la instrucción. Una vez concluida esta fase volvemos a la fase de búsqueda de la siguiente instrucción.

Antes de entrar en este bucle de ejecución de instrucciones hay que inicializar el computador para ejecutar un determinado programa. Para ello el sistema operativo debe:

1. Cargar el programa y los datos en memoria
2. Inicializar el PC para que apunte a la primera instrucción del programa a ejecutar.

Llegados a este punto podemos formular las siguientes definiciones:

Ciclo de instrucción. Tiempo necesario para leer y ejecutar una instrucción. El ciclo de instrucción es mayor para instrucciones más lentas y menor para instrucciones más rápidas. Normalmente el ciclo de instrucción se mide en ciclos de reloj. Por ejemplo, en el 8086 de Intel la instrucción ADD necesita 3 ciclos de reloj para completarse; la instrucción MUL necesita de 70 a 77 ciclos de reloj.

Ciclo máquina. Subperiodos en que se divide un ciclo de instrucción. Estos subperiodos pueden necesitar también uno o varios ciclos de reloj. En el tema siguiente veremos en detalle en qué consiste el ciclo máquina.

3.3.1. Modos de secuenciamiento

Nos referiremos aquí a la secuencia en que se pueden ejecutar las instrucciones dentro de un programa. Básicamente podemos distinguir cuatro modos:

1. **Ejecución secuencial:** Las instrucciones se ejecutan de forma secuencial incrementándose el PC sin discontinuidades. Es decir, se ejecutan en orden las instrucciones almacenadas consecutivamente en memoria.
2. **Salto** (transferencia de control): Mediante las instrucciones de salto el PC se puede cargar con una dirección distinta a la siguiente. De esta

forma se transfiere el control a otro punto del programa. El valor previo del PC (antes del salto) se pierde al modificar el contenido de éste.

3. **Subrutina** (transferencia de control): El valor del PC se modifica de forma que se provoca una transferencia de control a otro punto del programa (donde comienza una subrutina). La diferencia con el **salto** estriba en que el valor del PC (antes de su modificación) se "salva" en la pila. Para llamar a la subrutina se utiliza la instrucción **call**. La subrutina termina con una instrucción **ret** (de *return*) de forma que se recupera el valor del PC almacenado en la pila y la ejecución sigue por la instrucción siguiente al **call**.
4. **Interrupción** (transferencia de control): Mediante este modo también se produce una transferencia de control a una subrutina (llamada ahora subrutina de tratamiento de interrupción), pero, en general, dicha transferencia no la provoca el programador, sino que puede ocurrir en cualquier momento de la ejecución del programa. Por tanto, en la pila no debemos salvar únicamente el PC, sino también el resto del "estado de la máquina" (registro de estados, registros de propósito general, etc.). Al terminar la rutina de tratamiento de interrupción recuperamos el PC y el estado de la máquina y continúa la ejecución del programa.

3.3.2. Técnica pipeline de instrucciones

La técnica de segmentación o pipeline consiste en dividir una tarea en subtareas y ejecutar cada una de estas subtareas mediante una unidad funcional independiente. Por ejemplo, la tarea de fabricación de un coche se puede dividir en varias subtareas y ejecutar cada una de ellas por un operario dentro de una cadena de montaje. Esta estrategia se puede aplicar dentro de un procesador para acelerar la ejecución de instrucciones.

Por ejemplo, el 8086 de Intel, divide la tarea de ejecutar una instrucción en dos subtareas:

1. Búsqueda de instrucción
2. Decodificación, cálculo de EA, búsqueda de op. y ejecución

El 8086 dedica una unidad llamada BIU (unidad de interfaz del bus) a la búsqueda de instrucciones y otra unidad independiente, EU (unidad de ejecución), a la ejecución de instrucciones. Las dos unidades operan en paralelo, de forma que mientras se ejecuta la instrucción actual, ya se está buscando la instrucción siguiente en paralelo. De esta forma conseguimos reducir el tiempo de ejecución. Este caso particular de pipeline de dos etapas se conoce con el nombre de *prefetch* (prebúsqueda).

En general, podemos utilizar tantas unidades funcionales como subtareas queramos ejecutar. Por ejemplo, podríamos haber dedicado una unidad funcional para cada una de las 5 fases: búsqueda de instrucción, decodificación, cálculo de la EA, búsqueda de op. y ejecución; de forma que, en el caso ideal, el tiempo de ejecución de un programa se divide por 5.

3.4. EJEMPLO: MICROPROCESADOR 8086

3.4.1. Características generales

El 8086 y 8088 son microprocesadores de Intel de propósito general. Ambos son prácticamente idénticos excepto por el tamaño de su bus de datos externo. En el caso del primero, el bus de datos es de 16 bits, mientras que en el segundo, es de 8 bits. La razón para crear el 8088 con este bus de datos reducido fue prever la continuidad entre el 8086 y los antiguos procesadores de Intel (8085). En este ejemplo nos centraremos en el 8086.

Resumen de las características:

- Microprocesador de 16 bits (ancho del bus de datos)
- Bus de direcciones de 20 bits (máxima memoria direccionable: 1 MegaByte).
- E/S no mapeada en memoria. El espacio de direcciones de E/S es independiente del de memoria. Tamaño del espacio de E/S es de 64Kbytes
- Encapsulado de 40 pines. Algunas señales multiplexadas.
- Segmentación. Dos etapas: B.I.U (unidad de interfaz con el bus), que realiza la búsqueda *(prefetch)*, y E.U (unidad de ejecución), que recoge las instrucciones de la cola de la B.I.U. y las ejecuta.

3.4.2. Registros

El 8086 presenta un conjunto de 14 registros de 16 bits que se pueden agrupar de la siguiente forma:

Registros generales : Su función es el almacenamiento temporal de datos.

AX (acumulador). Es utilizado en las instrucciones aritméticas.
BX (base). Se usa generalmente para indicar un desplazamiento.
CX (contador). Se utiliza en bucles.
DX (datos). Se utiliza también en operaciones aritméticas.

Estos registros ofrecen la posibilidad de usarse además como registros de 8 bits, refiriéndose al byte más significativo con AH, BH, CH y DH o al menos significativo con AL, BL, CL y DL respectivamente.

Registros de segmento : Contienen la dirección de comienzo de los segmentos de memoria que conforman un programa.

CS o registro de segmento de código. Contiene la dirección del segmento donde están las instrucciones del programa.

DS o registro de segmento de datos. Hace referencia al comienzo del segmento de datos.

SS o registro de segmento de pila.

ES o registro del segmento extra. Se trata de un segmento que es utilizado para efectuar transferencias entre los tres segmentos principales.

Punteros : Se utilizan para designar el desplazamiento actual dentro de un segmento.

SP Puntero de pila.

BP Puntero base.

IP Puntero de instrucciones. Contiene el desplazamiento de la instrucción siguiente a ejecutar respecto al segmento de código del programa en ejecución.

Registros de índice : Se utilizan como desplazamiento relativo a un campo de datos.

SI Índice fuente.

DI Índice destino.

Registro de banderas (flags) : Sólo 9 de los 16 bits son significativos.

Los flags registran el estado del procesador después de una operación.

CF (Carry). Indica acarreo en las operaciones aritméticas.

OF (Overflow). Desbordamiento aritmético.

ZF (Zero). Se activa cuando el resultado de una operación es cero.

PF (Parity). Número par de bits.

SF (Sign). Bit de signo del resultado.

AF (Auxiliar). Relativo al uso de números en BCD.

Las banderas de control son tres. Registran el modo de funcionamiento del procesador.

DF (Direction). Controla la dirección (hacia delante o atrás) en las operaciones con cadenas de caracteres, incrementando o decrementando SI y DI.

IF (Interrupt). Controla la habilitación de las interrupciones.

TF (Trap). Controla la operación modo paso a paso.

3.4.3. Gestión de memoria

El bus de direcciones del 8086 es de 20 bits. Esto supone que el espacio de direccionamiento disponible es de $2^{20} = 1$ Mbyte. Cada una de estas posiciones de memoria se puede referenciar mediante un conjunto de 20 bits. Los registros internos de que dispone el microprocesador son de 16 bits, por lo que es necesario un pequeño truco para generar la dirección de 20 bits a partir de dos de estos registros. Esto se consigue introduciendo el concepto de *segmentación*. El espacio de direccionamiento del microprocesador se divide en bloques de 64 Kb, cada uno denominado segmento, de forma que, para hacer referencia a una posición de memoria determinada, hay que expresar el segmento en el que se encuentra y el desplazamiento que ocupa dentro de ese segmento. A esta forma de referenciar las posiciones de memoria se denomina direccionamiento lógico. De esta forma, una dirección lógica está formada por dos números, el primero expresa el segmento y el segundo el desplazamiento.

Dirección lógica = Segmento: Desplazamiento

El segmento hay que indicarlo siempre en un registro de segmento (CS, DS, etc..) y el desplazamiento puede expresarse de distintas formas, dando lugar a los distintos modos de direccionamiento. Como vemos, la dirección física de 20 bits (5 dígitos hexadecimal) se representa como dos enteros de 16 bits (4 dígitos hexadecimal). Por convenio, segmento y desplazamiento son números de 16 bits positivos.

Mediante este sistema es posible direccionar 16 * 64Kb = 1024 Kb = 1Mb posiciones de memoria. La conversión de esa dirección lógica a dirección física se realiza desplazando el registro de segmento 4 bits a la izquierda y sumándole el contenido del registro de desplazamiento.

*Dirección física = Segmento * 10h + Desplazamiento*

Se observa que el mapeo de direcciones físicas en direcciones lógicas no es biunívoco.

Esquema de almacenamiento

Este procesador sigue la regla "el byte menos significativo ocupa la posición más baja" (*Little Endian*). Así, si escribimos un dato en una posición de memoria, dependiendo si es byte, word, doble word,... se ubica en memoria según se esquematiza en la figura 3.4. La dirección de un dato es la de su byte me-

nos significativo. La memoria siempre se referencia a nivel de byte, es decir, si decimos la posición N nos estamos refiriendo al byte N-ésimo de memoria, aunque se escriba una palabra, doble palabra,...

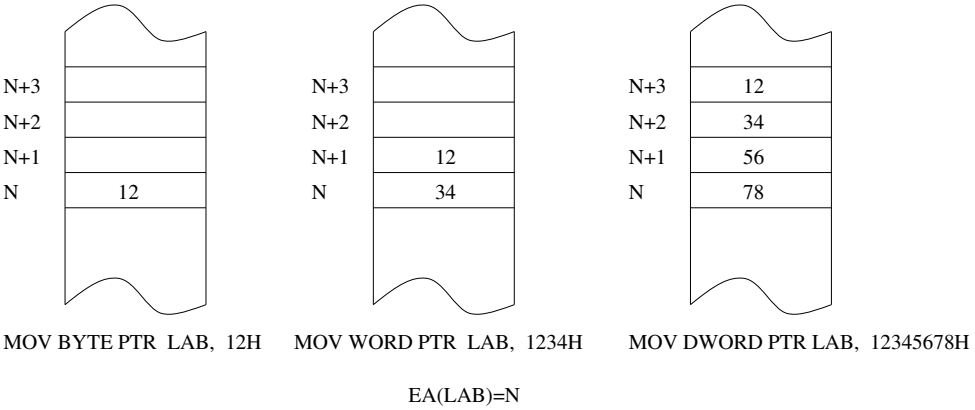


Figura 3.4: Ubicación de datos en memoria

Modos de direccionamiento

- *Direccionamiento inmediato:* `MOV AX,100H`
- *Direccionamiento directo a registro:* `MOV AX,BX`
- *Direccionamiento directo a memoria:* `MOV AX, [LABEL]`
- *Direccionamientos indirectos a registro*:
  - *Direccionamiento relativo a registro índice (SI,DI):* `MOV AX,[SI]` ; `MOV AX,[SI-3]`
  - *Direccionamiento relativo a registro base (BX,BP):* `MOV AX,[BP+2]`
  - *Direccionamiento base-índice con/sin desplazamiento:* `MOV AX,5[BX+DI]; MOV AX,LABEL[BX][DI]; MOV AX,LABEL[BP+DI+5]`
  - *Direccionamiento de pila:* `PUSH AX; POP AX`

Nota: En los modos de direccionamiento relativo a registro, el contenido del registro es el *desplazamiento* de la dirección de memoria. El segmento de la posición que está referenciando es por defecto DS, salvo que el registro base sea BP en cuyo caso el segmento es el contenido de SS. También se puede indicar explicitamente el semento, por ejemplo: `MOV AX, SS:[BP]`

3.4.4. Sistema de interrupciones

Dicho sistema emplea interrupciones *vectorizadas* que se pueden activar bien por hardware o software. Existen 256 interrupciones posibles ($INT0...INT255$). Cada interrupción tiene asociado un vector formado por 4 bytes, que recoge la dirección de memoria donde se encuentra la rutina de tratamiento de dicha interrupción (`CS:IP`). Estos vectores de interrupción forman una tabla que se encuentra en la parte más baja de la memoria, al comienzo de la RAM. En total existen 256 vectores de interrupción, de forma que la tabla de vectores ocupa 256×4 = 1Kb de memoria. La disposición física del vector de interrupción de la interrupción N es: $[N \times 4+3, N \times 4+2]$ =`CS`, $[N \times 4+1, N \times 4]$ =`IP`, donde `CS:IP` es la dirección de la rutina de atención a la interrupción. (Con $[K+1, K]$ denotamos el contenido de la palabra situada en la posición K de memoria). Como se acaba de ver, el vector de interrupción de la $INTN$, es una doble palabra situada en la posición física de memoria $N \times 4$.

Las fuentes de interrupción en este procesador son:

- **Líneas de interrupción (hardware): INTR** (interrupción enmascarable) y **NMI** (interrupción no enmascarable). La interrupción no enmascarable da lugar a la ejecución de la interrupción $INT2$. Si se quieren atender más de una interrupción enmascarable (teclado, ratón, ...), se ha de usar un controlador de interrupciones (PIC), por ejemplo el 8259.
- **Interrupción software:** Invocada desde el programa con la instrucción `INT N`, siendo N el número de la interrrupción a ejecutar.

El mecanismo de atención a una interrupción sigue el siguiente esquema:

1. Cuando se activa la interrupción, se debe guardar en la pila la dirección de la instrucción que está ejecutando la CPU, para poder volver después a ella. También se debe guardar el registro de estado, por si la rutina modifica alguna de las banderas.
2. Se busca en la tabla de vectores de interrupción la dirección donde se encuentra la rutina que se debe ejecutar.
3. Se carga `CS:IP` con el contenido del vector de interrupción asociado.
4. Se ejecuta la rutina de atención a la interrupción.
5. La ultima instrucción de la rutina es `IRET`, que se encarga de restaurar desde la pila `CS`, `IP` y el registro de banderas.
6. Se continúa ejecutando el programa por la instrucción siguiente a la que se interrumpió.

3.4.5. Ensamblador del 8086

La principal característica de un módulo fuente en ensamblador es que existe una clara separación entre las instrucciones y los datos. De hecho, un modulo fuente está formado por tres bloques independientes que se van a ubicar en tres segmentos distintos en la memoria RAM.

La estructura más general de un módulo fuente es:

* **Segmento de datos.** En este bloque se definen todas las variables que utiliza el programa con el objeto de reservar memoria para contener los valores asignados.
* **Segmento de código.** Este segmento incluye todas las instrucciones que forman el programa. En un mismo módulo fuente pueden existir más de un segmento de código.
* **Segmento de pila.** La pila es un tipo de memoria auxiliar donde se almacenan datos temporalmente. Es una memoria de tipo LIFO y existe un puntero que direcciona la cima de la pila. El segmento de pila indica la dirección de memoria donde se posiciona la pila y reserva un número determinado de posiciones consecutivas en memoria.

De estos tres segmentos, el único que obligatoriamente debe existir es el segmento de código.

3.4.6. La pila

La pila de un programa ensamblador se define con la directiva `SEGMENT STACK`. Si $N+1$ es la longitud en bytes reservada para la pila, la zona de pila es la comprendida entre la direcciones: `SS:0000H` y `SS:N`. De esta manera, `SP` se inicializa al valor $N+1$. El convenio es que `SP` apunte a la última palabra (posición más baja) que se ha introducido en la pila. Las operaciones de pila son siempre a nivel de palabra (2 bytes). Se muestra a continuación el efecto de las instrucciones más importantes que trabajan con la pila:

Instrucción	Operación de pila
PUSH AX	SP=SP-2; [SS:SP]=AL; [SS:SP+1]=AH;
POP AX	AL=[SS:SP]; AH=[SS:SP+1]; SP=SP+2;
CALL FAR PTR LABEL	SP=SP-2; [SS:SP]=CS(L); [SS:SP+1]=CS(H); SP=SP-2; [SS:SP]=IP(L); [SS:SP+1]=IP(H);
CALL NEAR PTR LABEL	SP=SP-2; [SS:SP]=IP(L); [SS:SP+1]=IP(H);
RET (lejano)	IP(L)=[SS:SP]; IP(H)=[SS:SP+1]; SP=SP+2; CS(H)=[SS:SP]; CS(L)=[SS:SP+1]; SP=SP+2
RET (cercano)	IP(L)=[SS:SP]; IP(H)=[SS:SP+1]; SP=SP+2;

3.4.7. Módulo fuente

Un **módulo fuente** está formado por instrucciones y directivas. Las instrucciones son representaciones simbólicas del juego de instrucciones del procesador (código máquina). Las directivas indican al ensamblador cierta manera de proceder a la hora de compilar, asi como para definir segmentos y reserva de espacio en memoria (variables, pila...). Las instrucciones se aplican en tiempo de ejecución mientras que las directivas se aplican en tiempo de ensamblaje.

Una **constante** es un nombre simbólico al que se le asocia un valor, el cual permanece invariable en el proceso de ejecución del programa. El ensamblador permite varias formas de representación de las constantes. Los valores numéricos pueden expresarse en base decimal (D), binaria (B), Hexadecimal (H) y octal (O). Por defecto, la representación se entiende que es decimal. Así, `10, 10D, 1010B, 0AH, 12O` representan al mismo valor numérico. Los caracteres alfanuméricos se representan encerrados entre comillas o bien expresando su código ASCII.

Una **etiqueta** es un nombre simbólico que acompaña a una instrucción o a una directiva en el módulo fuente y equivale al valor de la dirección de memoria donde se encuentra esa instrucción, es decir, la etiqueta "apunta" a esa posición.

Instrucciones

En el programa ensamblador, las instrucciones del 8086 aparecen con el formato general:

```
Etiqueta: Nombre_Instrucción Operando(s) ;Comentario
```

De estos campos, sólo el nombre de la instrucción es obligatorio. En la sintaxis del ensamblador cada instrucción ocupa una línea. Los campos se separan entre sí por al menos un carácter espacio (ASCII 32) y no existe distinción entre mayúsculas y minúsculas.

El campo *etiqueta*, si aparece, debe estar formado por una cadena alfanumérica seguida de un identificativo que expresa su atributo. La cadena no debe comenzar con un dígito y no se puede utilizar como cadena alguna palabra reservada del ensamblador ni nombre de registro del microprocesador.

El campo *nombre_instrucción* es un mnemónico de la instrucción del procesador. Está formado por caracteres alfabéticos (entre 2 y 6).

E1 campo *operando* indica dónde se encuentran los datos. Puede haber 0, 1 ó 2 operandos en una instrucción. Si hay 2, al primero se le denomina destino y al segundo fuente y deben ir separados por una coma. Los operandos pueden

ser registros, dirección de memoria o bien valores inmediatos. En cualquiera de los casos el tamaño debe ser byte o palabra.

E1 campo *comentario* es opcional y debe comenzar con el carácter `;`. También se puede especificar una línea completa de comentario si comienza por `;`.

Conjunto de instrucciones

- *Instrucciones de transferencia de datos.* Mueven información entre registros y posiciones de memoria o puertos de E/S. Pertenecen a este grupo las siguientes instrucciones: `MOV, LEA, IN, OUT, POP, PUSH, XCHG`:
  1. MOV AX, [VAR1]: (AX ← contenido de la variable VAR1)
  2. LEA AX, VAR1: (AX ← dirección de la variable VAR1)
  3. IN y OUT: Lectura y escritura en los puertos de E/S
  4. PUSH y POP: Escritura y lectura en pila
  5. XCHG AX, BX: (AX ↔ BX)
- *Instrucciones aritméticas.* Realizan operaciones aritméticas sobre números binarios o BCD. Son instrucciones de este grupo `ADD, SUB, CMP, INC, DEC, NEG, MUL, DIV`:
  1. ADD AX, BX: (AX ← AX+BX)
  2. SUB AX, BX: (AX ← AX-BX)
  3. CMP AX, BX: (AX-BX y actualiza los flags)
  4. INC AX: (AX ← AX+1)
  5. DEC AX: (AX ← AX-1)
  6. NEG AX: (AX ← C2[AX])
  7. MUL BL: (AX ← AL × BL)
  8. DIV BL: (AX ← AX/BL; AH=resto, AL=cociente)
- *Instrucciones lógicas.* Realizan operaciones de desplazamiento, rotación y lógicas sobre registros o posiciones de memoria. Están en este grupo las instrucciones: `AND, OR, XOR, NOT, SHL, SHR, SAL, SAR, ROL, ROR`:
  1. AND AX, BX: (AX ← AX$\wedge$ BX)
  2. OR AX, BX: (AX ← AX$\vee$ BX)
  3. XOR AX, BX: (AX ← AX$\oplus$ BX)
  4. NOT AX: (AX ← C1[AX])
  5. SHL AX,n / SHR AX,n: Desplazamiento lógico a la izquierda/derecha de n bits
  6. SAL AX,n / SAR AX,n: Desplazamiento aritmético a la izquierda/derecha de n bits. El desplazamiento aritmético a la derecha provoca extensión de signo

7. ROL AX,n / ROR AX,n: Rotación de n bits a la izquierda/derecha

- *Instrucciones de transferencia de control.* Se utilizan para controlar la secuencia de ejecución de las instrucciones del programa, tales como `JMP, JXX, JNXX, LOOP, CALL, RET`:
  1. JMP LABEL: Salto incondicional. Salta a la instrucción etiquetada con LABEL
     (IP ← LABEL)
  2. JXX y JNXX: Saltos condicionales:
     - JE/JNE: Salto si igual/no igual
     - JZ/JNZ: Salto si cero/no cero
     - JG/JL: Salto si mayor/menor
     - JGE/JLE: Salto si mayor-igual/menor-igual
     - JC/JNC: Salto si carry/no carry
     - JS/JNS: Salto si negativo/no negativo
     - JO/JNO: Salto si overflow/no overflow
  3. LOOP LABEL: Equivalente a "DEC CX" + "JNZ LABEL"
  4. CALL: Salto a subrutina
  5. RET: Retorno de Subrutina

Directivas

Las directivas son expresiones que aparecen en el módulo fuente e indican al compilador que realice determinadas tareas en el proceso de compilación. El uso de directivas es aplicable sólo al entorno del compilador, por tanto varían de un compilador a otro y para diferentes versiones de un mismo compilador. Las directivas más frecuentes en el ensamblador son:

- *Directivas de asignación:* Se utilizan para dar valores a las constantes o reservar posiciones de memoria para las variables (con un posible valor inicial). `DB y DW` son directivas que expresan al compilador que reserve memoria para las variables indicadas. Por ejemplo,

```
VAR1 DW 5
VAR2 DB 'A'
VAR3 DB 5 DUP (0)
VAR4 DW 2 DUP (5)
```

  La directiva `EQU` es utilizada para asignar un valor a una constante.

```
BASE EQU 25
```

- *Directivas de segmento:* La directiva `ASSUME` nos indica cuál es el registro de segmento que se usará para direccionar las etiquetas de un segmento lógico definido por `SEGMENT`. Por ejemplo:

```
ASSUME CS:CODIGO, DS:DATOS, SS:PILA
```

Para indicar el comienzo y fin de un segmento lógico en un módulo fuente se utilizan las directivas `SEGMENT` y `ENDS`. Un programa típico consta de las siguientes secciones:

```
DATOS SEGMENT
      ;definiciones de variables
      D1 DW 1
      D2 DB ?
DATOS ENDS
PILA  SEGMENT STACK
      ;reserva de espacio para pila
      DB 1024 DUP(0)
PILA ENDS
CODIGO SEGMENT
ASSUME CS:CODIGO, DS:DATOS, SS:PILA
      ;instrucciones del código
        ...
CODIGO ENDS
```

- *Directiva de Fin de programa:* La directiva `END` indica al compilador dónde termina el módulo fuente. Generalmente lleva a continuación la dirección de memoria en donde se encuentra la primera instrucción que se debe ejecutar.

```
END INICIO
```

La palabra inicio es alguna etiqueta o el nombre de algún procedimiento en el cual se encuentra la primera instrucción a ejecutar en nuestro programa.

- *Directivas de operando:* Se aplican a los datos en tiempo de compilación. Actúan sobre los datos para obtener información sobre ellos tales como `OFFSET`, que devuelve el desplazamiento de una variable con respecto al segmento en el que se encuentra. `SEG` devuelve el segmento en el que se encuentra una variable. La directiva `PTR` modifica momentáneamente el tamaño de una variable para una asignación concreta.

```
MOV BX, OFFSET VAR1  ; BX <- Dir. de VAR1
MOV AX, SEG VAR1     ; AX <- Segmento de VAR1
INC BYTE PTR [DI]    ; Considera el operando
                     ; apuntado por DI de tamaño BYTE
```

A título ilustrativo se muestra un programa fuente con vistas a ser ejecu-

tado en un entorno MS-DOS (las interrupciones se refieren a los servicios de este sistema operativo).

```
    CR      EQU 13  ; Retorno de carro
    LF      EQU 10  ; Llena una linea
;-------------------------------------------------
; Segmento de datos
    DATOS   SEGMENT
      D1  DB 00H, 4CH
      D2  DW 2H, 1234H
      D3  DD ?
      CADENA1  DB 'FUNDAMENTOS DE LOS COMPUTADORES',CR,LF,'$'
      CADENA2  DB 'Dpto. Arquitectura de Computadores',CR,LF,'$'
    DATOS   ENDS
;-------------------------------------------------
; Segmento de pila
    PILA    SEGMENT STACK
       DB 512 DUP ('?')
    PILA    ENDS
;-------------------------------------------------
; Segmentos de código
    CODIGO1 SEGMENT
              ASSUME CS:CODIGO1, SS:PILA, DS:DATOS
      INICIO: MOV AX,SEG DATOS
              MOV DS,AX
; Escribir texto
              CALL CERCANO
              CALL  FAR PTR ESCRIBIR
              MOV AX,[D1]
      FIN:    INT 21H     ; VOLVER AL DOS
      CERCANO PROC
              LEA DX,CADENA1
              RET
      CERCANO ENDP
    CODIGO1  ENDS
;-------------------------------------------------
    CODIGO2 SEGMENT
              ASSUME CS:CODIGO2
       ESCRIBIR PROC FAR
              PUSH AX  ; Salva el registro AX en la pila
              MOV AH,9H ; Funcion 9 del DOS
              INT 21H  ; Llamada a la interrupcion 21
              POP AX   ; Recupera registro AX
              RET      ; Vuelta al programa principal
       ESCRIBIR ENDP
    CODIGO2  ENDS
;-------------------------------------------------
       END  INICIO
```

SINOPSIS

En este tema hemos profundizado un poco más en el funcionamiento de un procesador. Hemos descrito los Registros, la Sección de Procesamiento y la Sección de Control, a modo de introducción para los dos siguientes temas. Por otro, lado hemos visto los conceptos de subrutina, interrupción y pipeline de instrucciones. Por último, se ha descrito un procesador comercial como el 8086 de Intel y hemos introducido así la programación en ensamblador.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

RELACIÓN DE PROBLEMAS

1. Para el número de 16 bits: 1001 0101 1100 0001 almacenado en el registro AX, muestra el efecto de:
   a) `SHR AX, 1`
   b) `SAR AX, 1`
   c) `SAL AX, 1`
   d) `ROL AX, 1`
   e) `ROR AX, 1`
2. ¿Cómo podrías poner a 0 el registro AX si no dispones de una instrucción de BORRAR (CLEAR)?
3. Inventa un método para intercambiar dos registros (AX y BX) sin usar un tercer registro o variable ni la instrucción XCHG. *Sugerencia:* piensa en la instrucción OR EXCLUSIVO.
4. Tenemos 250 números en C2 de 16 bits en las posiciones 500 a 998 de memoria. Escribir un programa en ensamblador del 8086 que cuente los números $P \geq 0$ y $N < 0$ y que almacene P en la posición 1000 de memoria y N en la 1002.
5. Tenemos 100 números positivos de 16 bits en las posiciones de memoria 1000 a 1198. Escribir un programa que determine cuál es el número mayor de todos y lo almacene en la posición 1200.
6. Escribir un procedimiento en lenguaje ensamblador del 8086 que calcule N! y lo almacene en la posición de memoria 502. Supondremos que 1<N<255, que el número N está en la posición 500 de memoria y que N! cabe en un registro de 8 bits.
7. Escribir una subrutina en lenguaje ensamblador que convierta un entero binario positivo (menor que 1000 y almacenado en la posición 2000 de memoria) a ASCII, guardando cada carácter a partir de la posición 3000 de memoria. Por ejemplo, el número 345 debe traducirse a los caracteres ASCII 3, 4 y 5 (33H, 34H y 35H en hexadecimal respectivamente)
8. a) Escribe el diagrama de flujo y el programa en ensamblador del 8086 de Intel para el algoritmo que realiza la siguiente función:
   Dada una tabla en la posición 100 de memoria con 50 números de 16 bits en complemento a dos, multiplicar por 2 los números pares y dividir por 2 los impares (división entera), dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar las instrucciones MUL y DIV. No considerar el caso de *overflow* al multiplicar por dos.

b) ¿Qué diferencia hay entre los desplazamientos lógicos y aritméticos a la derecha?

9. Escribe el diagrama de flujo y el programa en ensamblador del 8086 de Intel para el algoritmo que realiza la siguiente función:

   Dada una tabla en la posición 500 de memoria con 50 números de 16 bits en complemento a dos, forzar a cero los números negativos y multiplicar por tres los positivos, dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar ninguna instrucción de multiplicar (MUL). No considerar el caso de *overflow* al multiplicar por tres.

10. Suponga que el contenido de la memoria de un sistema basado en el i8086 es el que se muestra en la tabla 3.1. El valor de los registros es: DS=SS=E000H, SI=0001H, DI=0002H, BP=0003H, BX=0004H. Se define así mismo una etiqueta `TABLA`, que apunta a la dirección física E0001H y que pertenece al segmento de datos.

	posición física	contenido	
`MOV AX, [TABLA]`	.....	.....	
`MOV AX, [SI]`	E0006	DE H	
`MOV AL, [SI]`	E0005	BC H	
`MOV AX, [SI+2]`	E0004	9A H	
`MOV AX, TABLA[SI]`	E0003	78 H	
`MOV AL, TABLA[SI]`	E0002	56 H	
`MOV AH, TABLA[SI]`	E0001	34 H	<-- TABLA
`MOV AX, [BP+2]`	E0000	12 H	
`MOV AX, [BX+SI]`	.....	.....	
`MOV AX, [BP+SI]`	5D276	07 H	
`MOV AL, [BP+SI+1]`	5D275	06 H	
`MOV AH, TABLA[BX][SI]`	5D274	05 H	
`LEA AX, TABLA`	5D273	04 H	
`LEA DI, TABLA[SI+1]`	5D272	03 H	
`MOV AX, SEG TABLA`	5D271	02 H	
`MOV CX, OFFSET TABLA`	5D270	01 H	
	.....	.....	

Tabla 3.1: Programa que se analiza y contenido de la memoria

Para cada una de las instrucciones mostradas, determina como quedan afectados los registros correspondientes.

11. Sea el siguiente segmento de datos perteneciente a un programa escrito para el i8086:

```
      DATOS   SEGMENT
I     DB 'DEPARTAMENTO de ARQUITECTURA de COMPUTADORES'
II    DD 00200020H,1H
III   DQ 0B
IV    DB '1','2','3'
V     DW ?
      DATOS   ENDS
```

   a) Suponiendo que la primera posición física en memoria de dicho segmento es la 43210H, calcule las posiciónes físicas y lógicas de cada una de las etiquetas (variables) que componen el segmento.

   b) Haga un esquema de cómo quedan almacenados en memoria los datos.

12. El siguiente programa ensamblador para el i8086 fue ensamblado y ejecutado con éxito:

```
        LF EQU 10
        CR EQU 13
        A25 EQU 25h
        A26 EQU 26h
        LP EQU 512
DATOS   SEGMENT
        F       DW 25
        G       DW 80
        H       DD ?
        MUTEX   DB 0
DATOS   ENDS
PILA    SEGMENT STACK
        DB LP DUP (0)
PILA    ENDS
CODIGO1 SEGMENT
        ASSUME CS:CODIGO1, SS:PILA, DS:DATOS
INICIO:MOV AX,DATOS
        MOV DS,AX
        CALL FAR PTR FILL_INI
  WW :  CMP BYTE PTR [MUTEX],01H
        IN MUTEX,3F8H
        JNE WW
        MOV AX,F
        PUSH AX
        MOV AX,G
        PUSH AX
        CALL FAR PTR FILL_IN
  fin:  MOV AH,4CH
        INT 21H
```

```
CODIGO1  ENDS
CODIGO2 SEGMENT
       ASSUME CS:CODIGO2, DS:DATOS, SS:PILA
FILL_IN PROC FAR
       PUSH BP
       MOV BP,SP
       LEA SI,[BP+8]
       LEA DI,[BP+6]
       MOV AX,SS:[SI]
       MOV DH,AL
       MOV AX,SS:[DI]
       MOV DL,AL
       MOV AH,06H
       INT 10H
       POP BP
       RET
FILL_IN ENDP
FILL_INI PROC FAR
       MOV BYTE PTR H,33H
       RET
FILL_INI ENDP
CODIGO2 ENDS
       END INICIO
```

a) Muestre la evolución de la pila del programa indicando qué valores (o qué registros) van ocupando las posiciónes de ésta. (Al comenzar la ejecución del programa AX, BX, CX, DX, BP se suponen cero y SP vale la longitud de la pila)

b) Explique los modos de direccionamiento que aparecen en el programa.

c) Si las rutinas `FILL_IN, FILL_INI` hubieran estado en el mismo segmento que el programa principal, se podrían haber invocado como etiquetas cercanas (`CALL (NEAR PTR) ...`). ¿De qué manera afecta ésto a la ejecución del programa?

13. Las siguientes cuestiones están referidas al procesador Intel 8086:

a) Los dos fragmentos siguientes de programa ensamblador realizan una operación sobre los registros `AX, BX, CX`, cuyo resultado numérico es el mismo. ¿Cuál se espera que se ejecute en menos tiempo? ¿Por qué razones? ¿Qué instrucciones podrían ser eliminadas directamente sin modificar la semántica del código?

```
        PUSH AX                 MOV CL,CH
        PUSH BX                 MOV CH,BH
        ADD BH,AH               ADD CH,AH
        MOV CL,BH               CMP CL,CH
        POP BX                  JE  FIN
FIN:    POP AX          FIN:    ROL CX,08H
          (a)                     (b)
```

b) No obstante ambos fragmentos no modifican de igual forma el indicador de acarreo final `CF`. Suponiendo que AX=F0A0H, BX=0F70H, CX=8001H, antes de la ejecución, calcula cuánto vale `CF` y los registros (en hexadecimal) `AX, BX, CX` al terminar la ejecución de los dos códigos. Suponemos que sólo las instrucciones ROL y ADD afectan potencialmente al bit de acarreo, en su ejecución.[5] (Justificar resultados)

AX=
BX=
CX=

CF(a)=
CF(b)=

[5] Recuerda que el 8086 realiza las operaciones aritméticas en complemento a dos y debes usar el convenio del fabricante para rotaciones y desplazamientos si procede.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

4 Sección de Control

OBJETIVOS

- Presentar una técnica para el diseño de una Sección de Control micro-programada
- Aprender a codificar microsubrutinas asociadas a instrucciones máquina

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

4.1. INTRODUCCIÓN. CONTROL CABLEADO VERSUS MICROPROGRAMADO

Como se ha visto en el capítulo 3, la función de una sección de control es generar las señales necesarias para el funcionamiento del sistema en el estado de control actual y para determinar cuál es el siguiente estado de control. Estas operaciones las realiza en función del código de operación (C.O.) y el modo de direccionamiento (MD) de la instrucción que hay en el registro de instrucción (RI), cargada previamente desde la memoria principal.

Normalmente, la ejecución de una instrucción simple causa más de un cambio de estado en el sistema computador, enviándose, para cada uno de ellos, un grupo de señales de control a las unidades del sistema. Cada grupo de señales causa la ejecución de unas operaciones básicas específicas (denominadas *microoperaciones*), tal como la transferencia entre dos registros, el desplazamiento de los contenidos de un registro, o la selección de la función a realizar por la ALU. El siguiente grupo de señales de control puede o no depender de los resultados de la presente microoperación, o del estado de ciertos flags del sistema[6]. De esta manera, la interpretación y ejecución de una (macro)instrucción da lugar a una secuencia de operaciones máquina básicas (microoperaciones), cada una controlada por un grupo específico de señales de control (o microinstrucción).

La tarea principal de la unidad de control será secuenciar las microoperaciones, estableciendo en cada ciclo el estado de control actual (y siguiente) y generar las microinstrucciones precisas para la ejecución de la macroinstrucción. Por consiguiente, la sección de control debe contener la lógica necesaria para el almacenamiento de la información que identifica al estado de control actual y la lógica de decisión para la generación del identificador del estado de control siguiente. El identificador de estado actual puede estar conectado directamente a los puntos de control de la sección de procesamiento, o, más habitualmente, ser transformado mediante una lógica de decodificación en una plantilla que activa las señales de control.

La unidad de control cableada, cuyo diagrama básico de bloques se muestra en la figura 4.1, está compuesta por un conjunto de dispositivos de almacenamiento MSI (registros), para almacenar el estado de control actual y un conjunto de bloques combinacionales MSI (decodificadores) y SSI (puertas discretas), para generar el identificador de estado siguiente y las señales de

[6]En otras palabras, la siguiente microoperación será determinada en función de la microoperación actual y en algunos casos del estado de ciertos flags.

control. Típicamente, el diseño de este dispositivo se hace mediante la aplicación de heurísticos, de forma que, en principio, no existe una metodología precisa y concreta para su implementación. Una sección de control de este tipo suele ser (salvo para sistemas muy simples) poco flexible y estructurada, de forma que una modificación de su función requiere normalmente el rediseño parcial o total de la unidad.

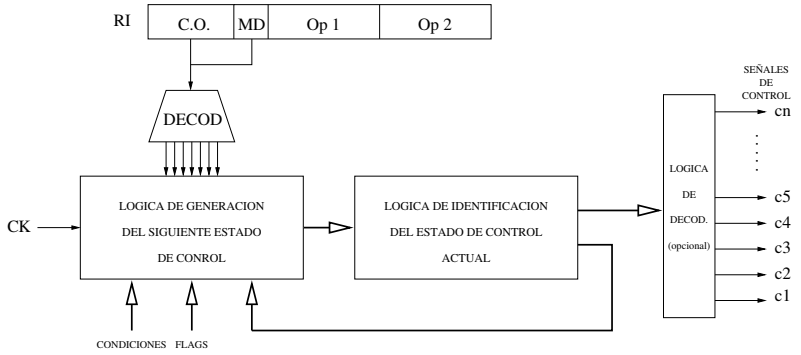


Figura 4.1: Unidad de control cableada

Frente a esta alternativa clásica, la unidad de control microprogramada, cuyo diagrama de bloques se muestra en la figura 4.2, utiliza un bloque de memoria LSI (ROM, PROM, PLA), llamado memoria microprograma.

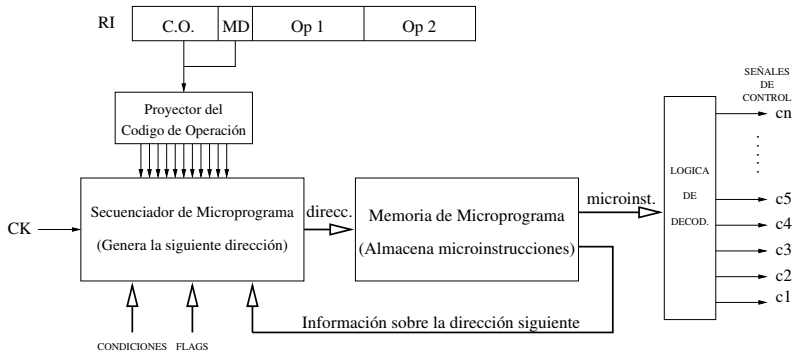


Figura 4.2: Unidad de control microprogramada

Esta memoria almacena la información del estado de control actual, que estará identificado por la dirección seleccionada en dicha memoria. El contenido de esta posición será la microinstrucción a ejecutar y suministra la informa-

ción necesaria para generar las señales de control y para que el secuenciador de microprograma determine la dirección de la siguiente microinstrucción a ejecutar.

El concepto de microprogramación fue propuesto en 1951 por Maurice Wilkes. Aunque conceptualmente la unidad de control microprogramada era mucho más flexible y estructurada que la cableada, la microprogramación no se empezó a usar comercialmente hasta 1964, con la serie System/360 de IBM. La razón es que los tiempos de acceso en las memorias ROM disponibles hasta entonces eran tan altos que la pérdida de prestaciones en los sistemas microprogramados era inaceptable para usos prácticos.

A partir de finales de los 60 y principios de los 70, el desarrollo de memorias de semiconductor rápidas y baratas permitió la expansión de la microprogramación a los minicomputadores, generalizándose su uso. Adicionalmente, el uso de memorias RAM para contener microprogramas posibilitó el diseño de sistemas microprogramables dinámicamente, en los que el programa de control puede ser cambiado durante el funcionamiento simplemente cambiando los contenidos de la memoria microprograma.

En la actualidad, el uso del control microprogramado es generalizado en los microprocesadores de juego de instrucción complejo (CISC). Sin embargo, en el diseño de procesadores de juego de instrucción reducido (RISC), en los que el control es relativamente simple, se prefiere el uso de unidades de control cableadas, con el fin de optimizar al máximo las prestaciones del hardware y reducir en lo posible el tiempo medio de ejecución de una instrucción (es decir, bajar el CPI – *ciclos de reloj por instrucción*–).

Puede resultar confuso y quizás poco didáctico intentar definir desde el primer momento los conceptos relativos a la microprogramación. Por ello, intentaremos hacer una primera aproximación a estos conceptos mediante un ejemplo para posteriormente adentrarnos más en detalle en temas avanzados. De acuerdo con esta filosofía, comenzaremos describiendo el flujo de datos de un hipotético procesador simple. A continuación, ilustraremos la técnica de control microprogramada mediante el desarrollo paso a paso de la sección de control microprogramada que controle a la sección de procesamiento inicial. Finalmente, se introducirá un conjunto de conceptos más complejos, relacionados con la temporización, optimización del diseño, y generación de microsubrutinas asociadas a instrucciones máquina.

4.2. DISEÑO DE UNA UNIDAD DE CONTROL MICROPROGRAMADA

Se desea diseñar una unidad de control microprogramada para la sección de procesamiento elemental de la figura 4.3. Esta sección está compuesta por registros de 8 bits (DR, AC, IR, PC, MAR), una ALU con operandos y resultado de 8 bits, multiplexores y buses de 8 bits para interconectar estos elementos.

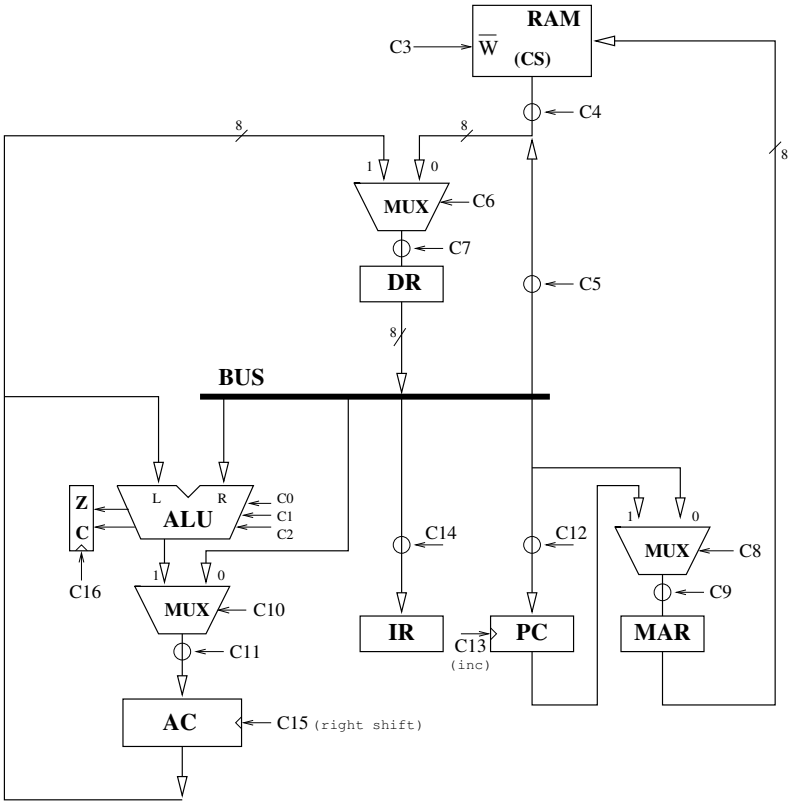


Figura 4.3: Una hipotética sección de procesamiento elemental

Es importante aclarar que aunque existe un bloque de memoria RAM en la figura, éste no pertenece a la sección de procesamiento (sino a la unidad de memoria del computador). En dicha memoria principal se almacenarán las instrucciones y los datos que procesará la sección de procesamiento.

Para simplificar el proceso de diseño, las entradas de control de los diversos componentes han sido abstraídas por conmutadores conceptuales, numerados

del 0 al 16 (c0-c16), y situados en las puertas de entrada y salida de los registros, estando cada conmutador controlado por una única línea de control. Un conmutador a la entrada de un registro la habilita cuando su línea de control pasa de 0 a 1 (es decir, es activo en flanco de subida) y permanece activado sólo un breve intervalo de tiempo, suficiente para que el dato sea almacenado en el registro. Un conmutador a la salida de un componente se activa cuando su línea de control pasa a valer 1 y permanece activado hasta que la línea vuelve a cero (es decir, es activo por nivel).

Aquellos registros que posean una salida conectada a un bus compartido poseerán capacidad tri-estado en dicha salida; si por el contrario el bus le está enteramente dedicado, la salida no poseerá esta capacidad.

El direccionamiento y acceso a memoria se realiza a través de los registros MAR y DR. El registro MAR posee una salida sin capacidad tri-estado, a través de la cual el contenido del registro MAR se usará como dirección a acceder en la memoria. Por su parte, la entrada del registro DR está controlada por un multiplexor: si la línea de control c6 está a 0, la entrada al registro DR será el contenido del bus de datos de la memoria; si, por el contrario, la línea c6 está a 1, el registro DR recibirá la información contenida en el registro AC. El registro MAR es denominado **registro de dirección de memoria**, mientras el registro DR se denomina **registro de dato de memoria**.

El punto de control c4 habilita el acceso a memoria. En lectura (c3=1), deja el dato en el bus de datos mientras c4 esta a uno (nivel). En escritura (c3=0), es en el flanco de bajada de c4 cuando se escribe el dato en la posición de memoria contenida en el registro MAR.

En la tabla 4.1 se sumariza la función de cada uno de los conmutadores del diseño.

Los diseños como los de la figura 4.3 son más un producto de la intuición y la experiencia que de la aplicación de un conjunto preestablecido de principios de diseño. Esto es, no existe una metodología exacta que nos diga que, para un conjunto de instrucciones dados y unos objetivos de rendimiento establecidos, el diseño de la sección de procesamiento es la elección óptima.

El siguiente paso del diseño consiste en conseguir que esta sección de procesamiento haga algo útil. Para ello, crearemos una sección de control microprogramada que, para cada instrucción del conjunto de instrucciones del procesador que estamos construyendo, genere una secuencia de conjuntos de señales que, ordenadamente, vayan activando los puntos de control pertinentes.

Comenzaremos por introducir un **registro de control** (o registro de microinstrucción –**MicroRI**–), de 17 bits de anchura, en el que cada bit esté co-

C	Significado	C	Significado
c0	Selección ADD en ALU	c1	Selección AND en ALU
c2	Selección COMP de la entrada L de la ALU (C1)	c3	Read(R) / Write($\overline{W}$)
c4	Habilitación memoria	c5	MEM ← DR
c6	0:(DR ← MEM), 1:(DR ← AC)	c7	Escritura en DR
c8	0:(MAR ← DR), 1:(MAR ← PC)	c9	Escritura en MAR
c10	0:(AC ← DR), 1:(AC ← ALU)	c11	Escritura en AC
c12	PC ← DR	c13	INC PC
c14	IR ← DR	c15	Desplaza a la derecha AC
c16	Carga registro de estado (C,Z)		

Tabla 4.1: Funciones de los conmutadores de control

nectado a una única línea de control, tal como se muestra en la figura 4.4.

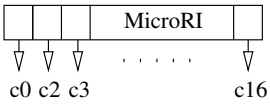


Figura 4.4: Registro de control

El problema ahora es el siguiente: colocar un valor en el registro de control, ¿desde dónde? Una segunda consideración a tener en cuenta es que el poner un único valor no será suficiente; realmente necesitaremos un conjunto de valores que varíe en el tiempo, de manera que una secuencia preestablecida de valores de control vaya apareciendo en el registro de control en intervalos periódicos.

Para conseguir esto, ya imaginamos las dos alternativas básicas:

- Diseño de un dispositivo secuencial (máquina de estados) formado por puertas lógicas, biestables, contadores, etc..., que dilucide en cada momento (en función del estado en el que se encuentre) qué valor debe almacenarse en el registro de control y a qué próximo estado debe cambiar. Esta solución es la que ya habíamos identificado en la sección anterior con el **control cableado**.
- Ya que tenemos que definir una secuencia de valores de 17 bits a almacenar en el MicroRI, ¿por qué no almacenamos estos valores en otra memoria (distinta de la memoria principal de la figura 4.3) y cada vez que necesitemos una palabra de control (microinstrucción) la leemos de dicha memoria? Esta solución conduce al diseño de una sección de **con-**

trol microprogramada como la de la figura 4.5.

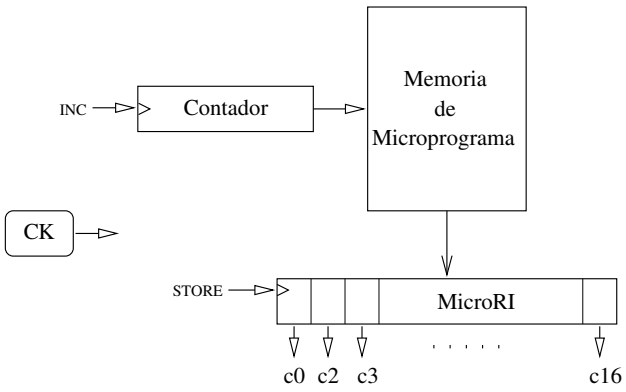


Figura 4.5: Sección de control microprogramado inicial

En la figura 4.6 vemos como el registro MicroRI se conecta a una **Memoria de Microprograma** (también llamada memoria de control, micromemoria o μMem.), direccionada por medio de un contador. El registro MicroRI tiene una señal *Store* (activa por flanco de subida). Por su parte, el contador tiene una entrada INC, activa en una transición de 0 a 1 (flanco de subida), que le hará incrementar en uno su valor actual.

Evidentemente, será también necesario un reloj, CK, que permita realizar la sincronización necesaria para ir leyendo periódicamente las microinstrucciones de la Memoria de Microprograma y almacenándolas en el MicroRI. Inicialmente se considerará un esquema de control muy simple, en el cual la señal de reloj se conectará directamente a la entrada *Store* del registro MicroRI y la misma señal invertida atacará el punto de control del registro Contador, como vemos en la figura 4.6. Por tanto, en cada flanco de subida del ciclo de reloj, el contenido de la memoria apuntado por el registro Contador se cargara en MicroRI y en el flanco de bajada se incrementará el Contador.

Llegados a este punto daremos unas cuantas definiciones:

- A este periodo de reloj se le denominará **tiempo del ciclo máquina** y será el intervalo entre conjuntos sucesivos de señales de control suministradas a la sección de procesamiento.
- Cada palabra de 17 bits de la memoria de control recibe el nombre de **microinstrucción**. Por consiguiente, una microinstrucción es un conjunto de señales de control o especificaciones que controla el flujo de datos y las funciones del dispositivo en la sección de procesamiento durante un

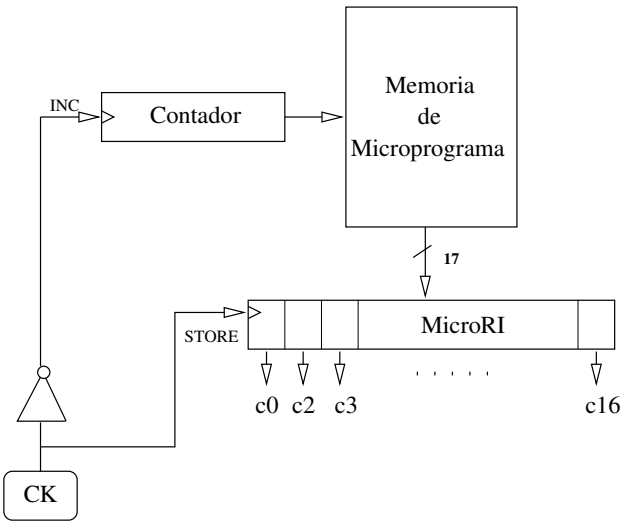


Figura 4.6: Sección de control microprogramado más detallada

ciclo máquina.

- La colección de microinstrucciones almacenadas en la memoria se denomina **microprograma**. La memoria que contiene el microprograma se denomina **memoria de microprograma**, mientras el contador que la direcciona actúa como un secuenciador de microprograma muy primitivo que también llamaremos **contador de microprograma** o **MicroPC**.

4.2.1. Características temporales

Un primer examen de la unidad de control diseñada, rápidamente revela que el sistema no funciona, debido a que no se ha tenido en cuenta la temporización en el uso de los recursos del procesador.

Un problema que se plantea es el hecho de que no hay ningún mecanismo que garantice que las señales de control asociadas a interruptores activos por flanco pasen a 0 tras haber tomado el valor 1 y, por tanto, que garantice la aparición del flanco de activación. Por ejemplo, si dos microinstrucciones sucesivas son del tipo:

```
XXXXX XXXX1 XXXXX XX
XXXXX XXXX1 XXXXX XX
```

el 1 de la segunda microinstrucción no tendrá ningún efecto. Esto es debido

a que ese punto de control, c9, se activa por flanco de subida, de forma que fuerza la carga del registro MAR cuando pasa de 0 a 1. Por tanto, la primera microinstrucción si tendrá el efecto esperado[7], pero la segunda no, ya que que en principio no existe ninguna transición baja-alta.

Es evidente, pues, que es necesario secuenciar la generación de diferentes conjuntos de señales de control dentro del ciclo máquina. Una solución simple que adoptaremos para el ejemplo que estamos desarrollando, en el que disponemos de una única señal de reloj, es emplear el flanco de bajada de ésta para temporizar las señales activas por flanco, de manera que si un bit del MicroRI asociado a un punto de control por flanco esté activo, se produzca una transición baja-alta con el flanco de bajada del reloj. Los cronogramas mostrados en 4.7 muestran la situación que deseamos implementar.

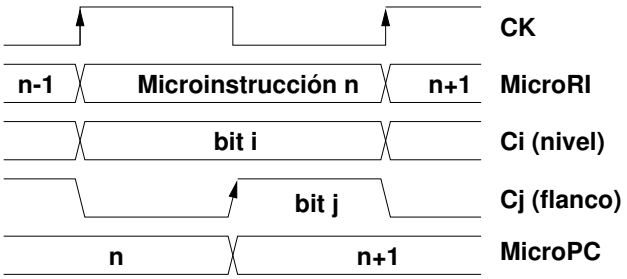


Figura 4.7: Formas de onda de las señales en el tiempo

Escribiendo ciertas ecuaciones lógicas para los puntos de control del sistema se puede diseñar un circuito que permita crear la secuencia descrita anteriormente. Ello se consigue fácilmente realizando una operación AND de los bits que corresponden a señales por flanco en MicroRI con $\overline{CK}$ tal como se esquematiza en la figura 4.8.

4.2.2. Ejemplos de microprograma

Una vez solucionada la temporización del ciclo máquina, ya es posible ejecutar en el procesador un pequeño microprograma. Veamos algunos ejemplos que nos servirán más adelante en la microprogramación del conjunto de instrucciones del procesador.

- $\boxed{AC \leftarrow Mem(DR)}$ Carga de AC con el contenido de la posición apuntada por DR. Para ello tenemos que llevar la dirección al registro MAR,

[7] siempre que la microinstrucción anterior tenga c9=0

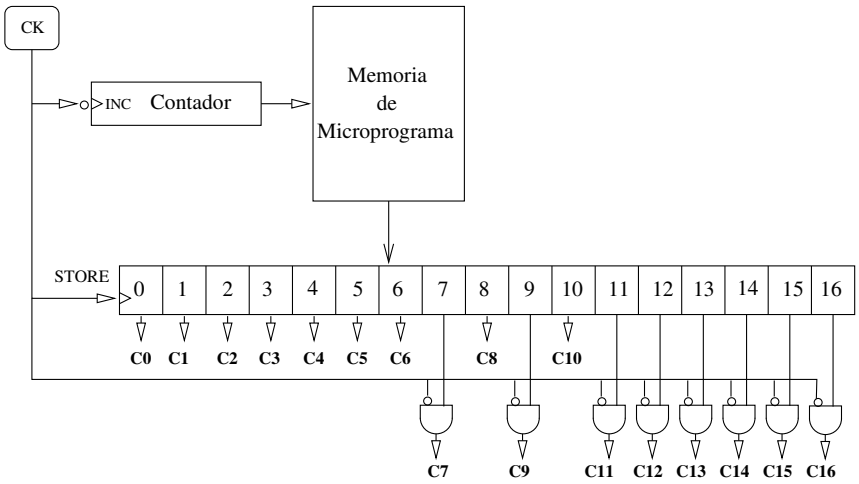


Figura 4.8: Esquema parcial de las conexiones de temporización

leer desde memoria en DR y de ahí al registro AC.

```
                    c0   c4   c8   c12  c16
1.MAR <- DR         0000 0000 0100 0000 0 (activos: c9)
2.DR  <- Mem(MAR)   0001 1001 0000 0000 0 (activos: c3,c4,c7)
3.AC  <- DR         0000 0000 0001 0000 0 (activos: c11)
```

- $\boxed{Mem(DR) \leftarrow AC}$ Cargar en la dirección de memoria apuntada por DR el contenido del acumulador AC. Como en el caso anterior la dirección destino se guardará en MAR, el acumulador se guarda en DR y DR se escribe en memoria.

```
                    c0   c4   c8   c12  c16
1.MAR <- DR         0000 0000 0100 0000 0  (activos: c9)
2.DR  <- AC         0000 0011 0000 0000 0  (activos: c6,c7)
3.Mem(MAR) <- DR    0000 1100 0000 0000 0  (activos: c4,c5)
```

- $\boxed{DR \leftarrow DR/2}$ Dividir el contenido de DR por dos. Para ello trasladamos el contenido de DR a AC, desplazamos a la derecha AC y el contenido de AC se volcará en DR.

```
                    c0   c4   c8   c12  c16
1.AC  <- DR         0000 0000 0001 0000 0  (activos: c11)
2.Shift_Right(AC)   0000 0000 0000 0001 0  (activos: c15)
3.DR  <- AC         0000 0011 0000 0000 0  (activos: c6,c7)
```

4.2.3. Capacidad de salto

Otro importante defecto del diseño, relacionado con el secuenciamiento de la unidad de control, es la falta de capacidad de ramificación en el microprograma, que obliga a pasar secuencialmente por todas y cada una de las microinstrucciones contenidas en la memoria de control. Un segundo problema de la unidad de control, es la ausencia de un mecanismo que permita tomar decisiones en el microprograma, es decir, dar saltos condicionados al valor de una o más variables de estado de la sección de procesamiento.

Ambos problemas expresan la necesidad de incluir algún procedimiento que permita la modificación, condicional o incondicional, del contenido del contador de microprograma. En una aproximación simple, suponiendo que la memoria de control tiene 64 palabras, se ampliará la microinstrucción con dos campos más. Uno de estos campos, al que se llamará **dirección de salto**, tiene 6 bits, y su función es contener la dirección de una palabra de la memoria de control. El otro campo, que llamaremos de **Condición de Salto** (o **CS**) tiene dos bits, con el siguiente significado:

00 **Ejecución secuencial** normal. El contador de secuencia se incrementa en 1 y la próxima microinstrucción a ejecutar es la inmediatamente siguiente a la actual.

11 **Salto incondicional**. La dirección de la siguiente microinstrucción a ejecutar está contenida en el campo dirección de salto de la microinstrucción en ejecución. El contador se cargará con el valor contenido en los 6 bits de este campo.

01 **Salto condicional** asociado al acarreo de salida **C** de la ALU. Si este acarreo de salida es 1, la microinstrucción siguiente viene definida por el campo de dirección de salto como en el salto incondicional. En caso contrario, la ejecución será secuencial (normal).

10 **Salto condicional** asociado a la salida **Z** de la ALU. Si el resultado a la salida de la ALU es 0 (línea Z a 1), la microinstrucción siguiente viene definida por el campo de dirección de salto de la actual microinstrucción como en el salto incondicional. En caso contrario, la ejecución será secuencial (normal).

Para implementar esta capacidad de salto (condicional o incondicional) de la sección de control podemos utilizar el hardware de la figura 4.9. Los bits 17 y 18 de la microinstrucción forman la microfunción de control de salto, mientras los bits 19 a 24 constituyen el campo de la dirección de salto.

Vemos como hemos ampliado la funcionalidad del Contador de microprograma (MicroPC a partir de ahora), de forma que además de ser incrementado,

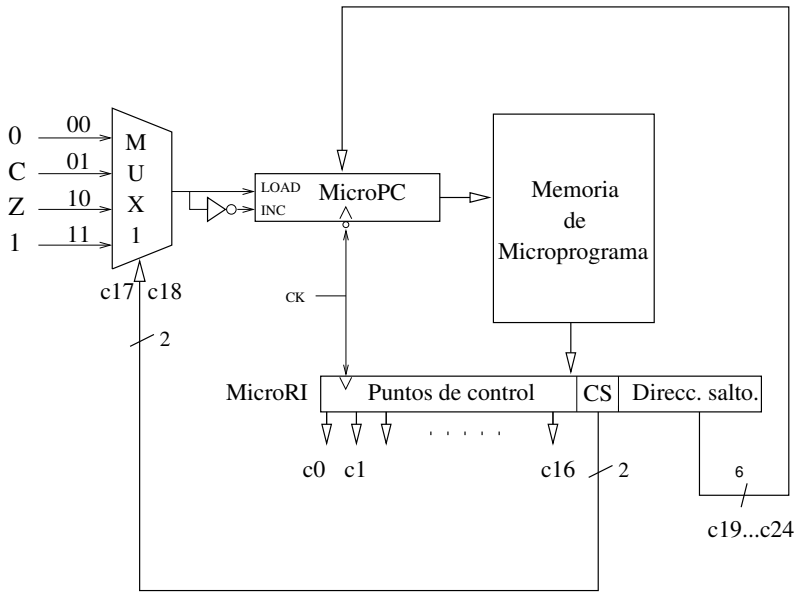


Figura 4.9: Sección de control con capacidad de salto

también puede ser cargado con un nuevo valor que llega desde el campo *Dirección* del MicroRI. El incremento o la carga del MicroPC estará determinado por las entradas de control *INC* o *LOAD* respectivamente, siendo estas entradas activas a nivel alto. El incremento del MicroPC está asociado a la ejecución secuencial, mientras que la carga implicará un *salto* a nivel de microprograma. Los dos bits del campo *Condición de Salto*, **CS**, seleccionan una de las entradas del **MUX1** de forma que:

CS=00 La entrada 00, alimentada con un 0 lógico, provocará el incremento del MicroPC. Es decir, ejecución secuencial.

CS=11 La entrada 11, alimentada con un 1 lógico, provocará la carga del MicroPC. Es decir, salto incondicional.

CS=01 La entrada 01, alimentada con el valor del flag C determinará la carga o el incremento del MicroPC según C=1 o C=0, respectivamente. Es decir: salto condicional al flag C.

CS=10 La entrada 10, alimentada con el valor del flag Z determinará la carga o el incremento del MicroPC según Z=1 o Z=0, respectivamente. Es decir: salto condicional al flag Z.

Este simple mecanismo de salto permite incrementar notablemente las po-

sibilidades del sistema. A cambio, se incrementa sustancialmente el tamaño de la microinstrucción. En este sentido, el campo de dirección de salto resulta especialmente costoso, pues sus 6 bits suponen un incremento de un 35 % en la longitud de la palabra de control, y sólo se emplea en unas pocas microinstrucciones.

Como ejemplo microprogramemos la operación `JZ DR`: saltar a la dirección apuntada por DR cuando Z=1, en caso contrario incrementar PC, esto es:
Si Z=1 $PC \leftarrow DR$ *si no* $PC \leftarrow PC + 1$

```
                                        microprograma
                        c0   c4   c8   c12  c16 c17 c19   c24
1.Si Z=1 goto 4         0000 0000 0000 0000 0   10  000 100
2.PC <- PC+1            0000 0000 0000 0100 0   00  000 000
3.goto 5                0000 0000 0000 0000 0   11  000 101
4.PC <- DR              0000 0000 0000 1000 0   00  000 000
5.siguiente
  microinstrucción ..................................
```

En este ejemplo las operaciones de las microinstrucciones 2 y 3 se podían haber realizado en una sola microinstrucción pero se ha hecho así por claridad.

4.2.4. Microprogramación del conjunto de instrucciones

Una vez diseñados el *datapath* y el secuenciador de control del procesador elemental, se podría proceder a la microprogramación de las diversas (macro)instrucciones de que se quiera dotar a su juego de instrucciones. Es decir, tenemos que completar el diseño teniendo en cuenta que al fin y al cabo el procesador que hemos diseñado tiene que ser capaz de ejecutar cualquier programa que tenga en su memoria principal. Por tanto debe ejecutar cualquier instrucción del conjunto de instrucciones de este procesador, a través de las etapas de *búsqueda de instrucción*, *decodificación*, *cálculo de la dirección efectiva*, *búsqueda de operandos* y *ejecución* como se explicó en el tema anterior. Más concretamente, la ejecución de cada instrucción se llevará a cabo mediante la ejecución de una **microsubrutina** (o **microrrutina**) asociada a cada instrucción máquina.

Se supone que antes de ejecutar el programa, el sistema operativo, habrá colocado el programa en memoria y habrá inicializado el contador de programa, PC, con la dirección de la primera instrucción máquina a ejecutar. A partir de ahí, la sección de control será la encargada de ir leyendo las instrucciones que sean apuntadas por el PC, cargándolas en el IR y ejecutándolas, hasta que el programa termine y se devuelva el control al S.O.

En cuanto al registro IR, vemos que no es más que un registro que se de-

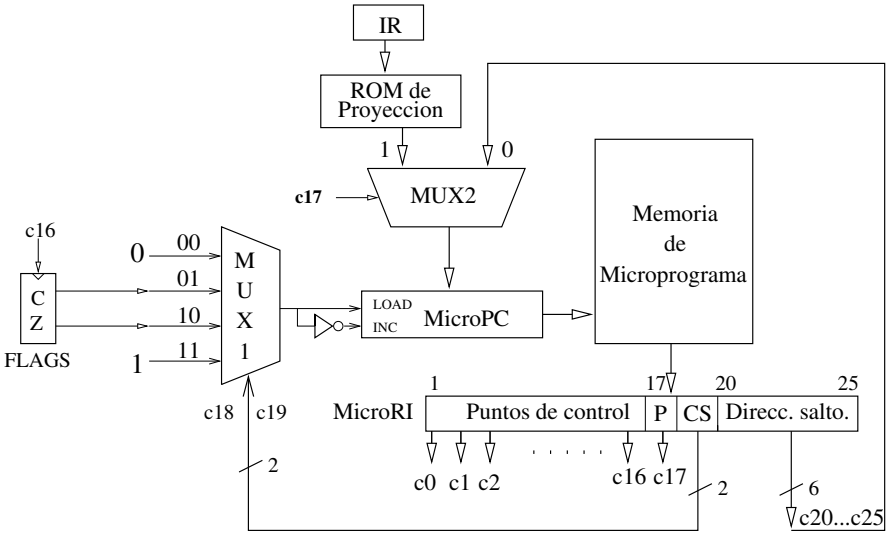


Figura 4.10: Sección de control completa

be cargar desde el BUS MEM, con la instrucción que ha seleccionado el PC en memoria. El código de operación de la instrucción cargada en el IR debe inicializar el MicroPC de forma que apunte a la primera microinstrucción de la microsubrutina asociada a dicho C.O. Para encontrar la dirección de una microrrutina a partir de su C.O. asociado, utilizaremos una tabla implementada mediante una ROM, (**ROM de Proyección**). Esta ROM contendrá en la dirección indicada por el C.O. la dirección de la microrrutina asociada a dicha macroinstrucción. Por ejemplo, si el código de operación de la instrucción ADD es 25 y tiene asociada una microrrutina que está en la dirección 97 de la Memoria de Microprograma, la posición 25 de la ROM de proyección debe contener un 97.

Dado que el MicroPC es cargado algunas veces con la dirección que viene de la ROM de Proyección y otras veces con la que viene del campo Dirección del MicroRI, utilizaremos el multiplexor MUX2 para seleccionar cuál de las direcciones será la que actualice el MicroPC. Ello implica añadir un nuevo punto de control que denominaremos **bit de proyección**. Por cuestiones de significado, añadimos este bit inmediatamente antes del campo de condición de salto, de manera que será c17, pasando c18, c19 a ser los puntos de control de salto y c20 a c25 los bits de dirección de salto. En la figura 4.10 se muestra la unidad de control microprogramada completa. En esta figura no mostramos,

por simplificación, la lógica adicional necesaria de temporización, pero no hay que olvidar que las microordenes son secuenciadas ordenadamente, tal y como se explicó en la sección 4.2.1.

Retomaremos ahora, una vez completado el diseño del procesador, el problema de escribir las microrrutinas asociadas a cada una de las instrucciones máquina. Cualquier microrrutina debe contener las microinstrucciones necesarias para realizar las siguientes funciones:

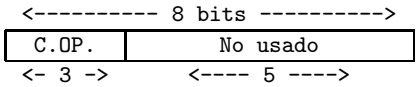
1. El contenido de la memoria apuntado por el PC debe ser cargado en el registro RI (**búsqueda de instrucción**). A continuación y a partir del C.O. recién cargado en el RI, se debe cargar la dirección inicial de la microrrutina asociada en el el registro MicroPC (**Decodificación**). Además, se incrementará el registro PC para que apunte a la siguiente palabra de memoria.
2. Es ejecutado el núcleo de la microrrutina: **cálculo de dirección efectiva**, **búsqueda de operando(s)** y **ejecución**.
3. La última microinstrucción de la microrrutina deberá saltar de nuevo al conjunto de microinstrucciones encargadas de la búsqueda y decodificación de instrucciones, que procederá a leer de la memoria principal la siguiente instrucción a ejecutar, volviendo así a comenzar el ciclo.

Dado que la fase de búsqueda y decodificación de instrucciones es común e independiente de la instrucción en particular que vaya a ser ejecutada, escribiremos las microinstrucciones encargadas de dicha misión una única vez en la memoria de microprograma, por ejemplo, a partir de la posición 0 de esta memoria. Llamaremos a ese conjunto de microinstrucciones, **microrrutina de búsqueda y decodificación**, o, **microrrutina de** *fetching*. Al resto de las fases (cálculo de dirección efectiva, búsqueda de operando y ejecución) lo llamaremos el **núcleo de la microrrutina** asociada a una instrucción dada.

Por tanto, cuando se va a ejecutar un programa y el S.O. lo carga en memoria e inicializa el PC, apuntando a la primera instrucción, también se debe resetear el MicroPC al valor cero. Una vez completada la inicialización, se ejecuta la microrrutina de *fetching* (recordemos que está almacenada a partir de la posición 0). Así, se carga la primera instrucción en el RI y en la decodificación, la ROM de Proyección proporciona la dirección del núcleo de la microrrutina asociada al código de operación que se encuentre en RI. Esa dirección se cargará en el MicroPC, o dicho de otra forma, estaremos saltando a la posición de la Memoria de Microprograma que contiene dicho núcleo de la microrrutina. Una vez completada ésta, la última microinstrucción debe contener un salto a la posición 0 para volver a buscar y ejecutar la siguiente

instrucción máquina.

Supondremos que en esta máquina hipotética, las instrucciones tiene un tamaño fijo de 8 bits, interpretándose de la siguiente manera: los 3 bits más significativos corresponden con el código de operación y los 5 menos significativos no se usan. Es decir:



Siguiendo las indicaciones marcadas en el apartado anterior, vamos a elaborar el microprograma que se almacenará en la micromemoria y que implementa el conjunto de instrucciones mostrado en la tabla 4.2.

Mnemónico	Descripción	C.OP.
LOAD [X]	AC ← Mem(X)	000
STORE [X]	Mem(X) ← AC	001
ADD [X]	Suma: AC ← AC + Mem(X)	010
AND [X]	AND lógico: AC ← AC ∧ Mem(X)	011
JMP	Salto incondicional: PC ← AC	100
JZ	Salta si cero: si Z=1 ⇒ PC ← AC	101
NOT	AC ← C1(AC)	110
SHR	AC ← 0,AC[7:1]	111

Tabla 4.2: Conjunto de instrucciones de la CPU

Vemos como en total tendremos 8 instrucciones distintas, donde las 4 primeras utilizan direccionamiento directo a memoria, siendo X una palabra de 8 bits que debe estar a continuación del C.O. en la memoria principal. Sin embargo las últimas cuatro instrucciones son de 0 direcciones y toman implícitamente el operando del registro AC.

El contenido de la micromemoria resultante para estas instrucciones se muestra en la tabla 4.3.

Sobre este microprograma cabe hacer los siguientes comentarios:

- La búsqueda de instrucciones se realiza en las microinstrucciones 0 a 3.
- Las instrucciones se hayan en: LOAD: líneas 4 a 8; STORE: líneas 9 a 13; ADD: líneas 14 a 18; AND: líneas 19 a 23; JMP: líneas 24 y 25; JZ: líneas 26 a 29; COMP: línea 30; SHIFT: línea 31.
- Cada instrucción acaba con la microinstrucción `goto 0`, que retorna a la búsqueda de instrucción.
- La decodificación se realiza en la línea 3, siendo ésta la única microinstrucción que activa el bit 17 (proyección).

```
Operación                N  Microprograma
                            c0 1    5    9    13   17 18 20    25
------------ búsqueda y dec. ------------------------------------
MAR <- PC                0   0 0000 0001 1000 0000 0  00 000 000
DR  <- Mem(MAR)          1   0 0011 0010 0000 0000 0  00 000 000
IR  <- DR                2   0 0000 0000 0000 0100 0  00 000 000
PC  <- PC+1              3   0 0000 0000 0000 1000 1  11 000 000
----------------- load X -------------------------------------
MAR <- PC                4   0 0000 0001 1000 0000 0  00 000 000
DR  <- Mem(MAR), PC++    5   0 0011 0010 0000 1000 0  00 000 000
MAR <- DR                6   0 0000 0000 1000 0000 0  00 000 000
DR  <- Mem(MAR)          7   0 0011 0010 0000 0000 0  00 000 000
AC  <- DR; goto 0        8   0 0000 0000 0010 0000 0  11 000 000
---------------- store X -------------------------------------
MAR <- PC                9   0 0000 0001 1000 0000 0  00 000 000
DR  <- Mem(MAR), PC++   10   0 0011 0010 0000 1000 0  00 000 000
MAR <- DR               11   0 0000 0000 1000 0000 0  00 000 000
DR  <- AC               12   0 0000 0110 0000 0000 0  00 000 000
Mem(MAR) <- DR; goto 0  13   0 0001 1000 0000 0000 0  11 000 000
----------------- add X --------------------------------------
MAR <- PC               14   0 0000 0001 1000 0000 0  00 000 000
DR  <- Mem(MAR), PC++   15   0 0011 0010 0000 1000 0  00 000 000
MAR <- DR               16   0 0000 0000 1000 0000 0  00 000 000
DR  <- Mem(MAR)         17   0 0011 0010 0000 0000 0  00 000 000
AC  <- AC+DR; goto 0    18   1 0000 0000 0110 0001 0  11 000 000
----------------- and X --------------------------------------
MAR <- PC               19   0 0000 0001 1000 0000 0  00 000 000
DR  <- Mem(MAR), PC++   20   0 0011 0010 0000 1000 0  00 000 000
MAR <- DR               21   0 0000 0000 1000 0000 0  00 000 000
DR  <- Mem(MAR)         22   0 0011 0010 0000 0000 0  00 000 000
AC  <- AC & DR; goto 0  23   0 1000 0000 0110 0001 0  11 000 000
------------------- jmp ---------------------------------------
DR <- AC                24   0 0000 0110 0000 0000 0  00 000 000
PC <- DR; goto 0        25   0 0000 0000 0001 0000 0  11 000 000
-------------------- jz ---------------------------------------
Si Z=1 goto 28          26   0 0000 0000 0000 0000 0  10 011 100
goto 0                  27   0 0000 0000 0000 0000 0  11 000 000
DR <- AC                28   0 0000 0110 0000 0000 0  00 000 000
PC <- DR; goto 0        29   0 0000 0000 0001 0000 0  11 000 000
------------------- not ---------------------------------------
AC <- NOT(AC) ; goto 0  30   0 0100 0000 0110 0001 0  11 000 000
------------------  shr ---------------------------------------
SH_RIGHT(AC); goto 0    31   0 0000 0000 0000 0010 0  11 000 000
```

Tabla 4.3: Contenido de la memoria de microprograma

Así pues, propuesto el contenido de la micromemoria, el contenido del sistema de proyección (la dirección está expresada en binario y la salida en decimal), que asocia a cada instrucción el comienzo de la microrutina que la ejecuta, viene dado por:

C.OP.	000	001	010	011	100	101	110	111
Línea	4	9	14	19	24	26	30	31

4.3. DISEÑO AVANZADO

En esta sección se propone un diseño avanzado para una sección de control asociada a la sección de datos que se muestra en la figura 4.11. Esta sección está compuesta por cuatro registros (I, A, B y D), un sumador binario completo, un desplazador y un multiplexor de dos entradas; adicionalmente, existen dos registros cuyo contenido es siempre cero. Todos estos componentes están conectados mediante un conjunto de buses. En particular el **BUS X** proporciona la entrada a los registros, el **BUS MEM.** es el bus de datos de la memoria RAM, el **BUS L** alimenta la entrada izquierda (*Left*) de la ALU, así como el **BUS R** hace lo propio con la derecha (*Right*). Aunque la anchura de los componentes y las vías de datos no es importante para el diseño de la unidad de control, supondremos una anchura de 16 bits. Las entradas de control de los dispositivos están sumarizadas en la tabla 4.4.

C	Significado	C	Significado
c1	I → BUS L del Sumador	c2	A → BUS L del Sumador
c3	B → BUS L del Sumador	c4	0 → BUS L del Sumador
c5	B → BUS R del Sumador	c6	D → BUS R del Sumador
c7	0 → BUS R del Sumador	c8	Sumador → Desplazador
c9	BUS X → I	c10	BUS X → A
c11	BUS X → B	c12	D-Mpx → D
c13	D → BUS MEM	c14	0 → pasa BUS X; 1 → pasa BUS MEM
c15	Carry de entrada al Sumador	c16	1 → Desplazamiento; 0 → No desplaza
c17	1 → Derecha; 0 → Izquierda	c18	Habilitación de Memoria
c19	1 → Lectura; 0 → Escritura		

Tabla 4.4: Funciones de los conmutadores de control

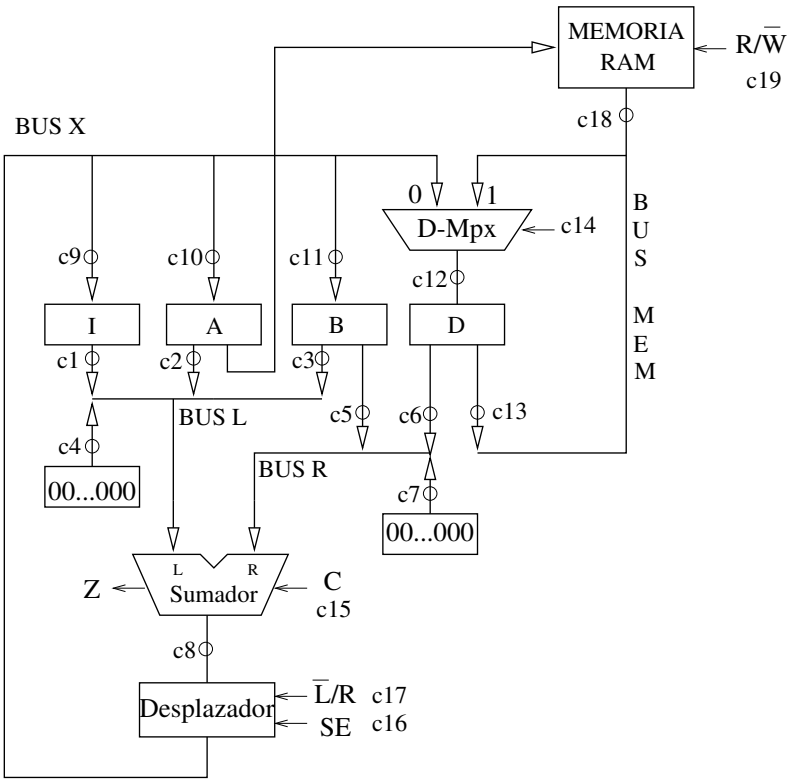


Figura 4.11: Sección de procesamiento elemental

4.3.1. Características temporales

En el diseño de la unidad de control propuesta en la sección 4.2 se temporizó las señales de control mediante un único reloj. Ello impone una gran limitación en cuanto a la ejecución de una microinstrucción, ya que el número de registros que puede "atravesar" un dato por cada periodo de reloj es uno, es decir, en cada periodo de reloj un dato puede ser transferido de un registro a otro nada más. Sin embargo, nuestro objetivo puede ser más ambicioso y podemos desear que en un periodo de reloj, esto es, en una sola microinstrucción, se pueda realizar la transferencia de un dato por un camino que implique más de un registro.

Podemos, para ello, dividir el ciclo máquina en una secuencia de subciclos, en los que se carguen una serie de registros, con lo que en un sólo ciclo podamos trasladar un dato, si quisiéramos, desde la memoria hasta la salida de la ALU.

Esta división podría ser:

Fase 1. Habilitación de la memoria de microprograma y lectura de la microinstrucción actual.

Fase 2. Carga del MicroRI con la microinstrucción actual.

Fase 3. Habilitación de la salida del MicroRI. Deshabilitación de la memoria de microprograma.

Fase 4. Si la microinstrucción actual lo requiere, habilitación de los controles de salida c1 a c7 y c13 y de la selección de memoria principal, c18.

Fase 5. Si la microinstrucción actual lo requiere, habilitación del conmutador c8 para pasar la salida del sumador al desplazador. Generación de una transición 0-1 en la entrada INC del contador de microprograma.

Fase 6. Deshabilitación de los conmutadores c1 a c7, c13 y c18. Si la microinstrucción lo requiere, habilitación del conmutador c16 para realizar un desplazamiento.

Fase 7. Si la microinstrucción actual lo requiere, habilitación de los conmutadores c9 a c12 para almacenar la salida del desplazador a través del bus X.

Las señales de control c14, c15, c17 y c19 son señales asíncronas (activas por nivel), pues no inician ninguna acción. En consecuencia, no requieren un control temporal, bastando con garantizar que tomen el valor deseado desde el momento en que se habilita la salida del registro MicroRI.

Una posible implementación de esta secuencia podría ser la generación de 7 pulsos de reloj dentro de cada periodo de 200 ns, para distinguir cada fase, pero eso implica subir la frecuencia del reloj. Sin embargo, es posible simplificar el circuito de reloj necesario usando una única señal de reloj para crear cuatro pulsos temporales (denominados T1 a T4), también de periodo 200 ns, pero desfasados del reloj principal en tal forma que se solapen.

Ésto se puede conseguir mediante una cadena de elementos de retardo (que pueden consistir simplemente en dos inversores en secuencia). Dado que la fase de acceso a la memoria de control es, presumiblemente, la más lenta, se tomará un retardo de 50 ns entre T1 y T2 y 23 ns de retardo entre T2 y T3 y entre T3 y T4. En la figura 4.12 se muestran las formas de onda resultantes. El solapamiento de las señales permite discriminar un considerable número de intervalos temporales, definidos simplemente mediante el producto (AND) y suma (OR) lógicos de las señales T1 a T4.

Las ecuaciones lógicas para los puntos de control del sistema deben implicar la secuencia de siete pasos, como se describirá más adelante. Así, según se vió en dicho esquema de fases, las acciones iniciales deben ser habilitar la

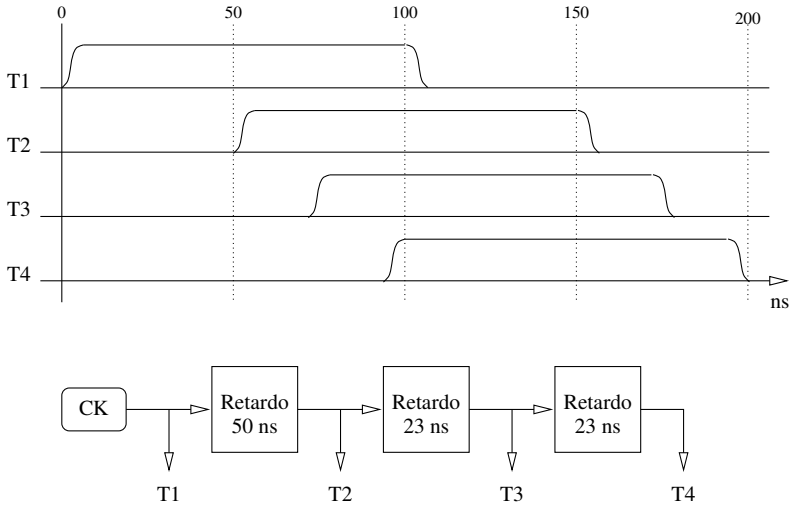


Figura 4.12: Formas de onda de las señales de tiempo

memoria de microprograma, cargar el MicroRI con la salida de esta memoria de microprograma, tras un tiempo igual al tiempo de acceso a dicha memoria y, por último, habilitar la salida del MicroRI y deshabilitar la lectura de la memoria de microprograma. La gestión de la memoria de microprograma puede hacerse simplemente conectando la señal $T1\ AND\ \overline{T2}$ como señal de control *Enable* de la memoria de microprograma.

Por su parte, si se supone que el tiempo de acceso de la memoria de microprograma es de 40 ns, la señal de habilitación de escritura en el MicroRI, *Store*, deberá conectarse a T2 y, por tanto, el registro MicroRI se cargará en el flanco de subida de T2. Por último, puesto que el MicroRI debe proporcionar las señales de control a la unidad de procesamiento desde ese instante hasta el final del ciclo máquina, se conectará la habilitación de lectura del MicroRI, *Enable*, a la señal T2 OR T4. En la figura 4.13 se representa parte del esquema de conexionado que permite el secuenciamiento descrito.

El control de los conmutadores c1 a c7, c13 y c18, necesario en las fases 4 y 6, se logra simplemente realizando el AND de las señales de control correspondientes con la señal T3. El conmutador c8, que debe abrirse en un instante posterior, se controla mediante la señal resultante del AND de la línea de control c8 y de la señal T4.

Por su parte, la operación de desplazamiento debe retardarse hasta que el registro de desplazamiento haya recibido la información procedente del su-

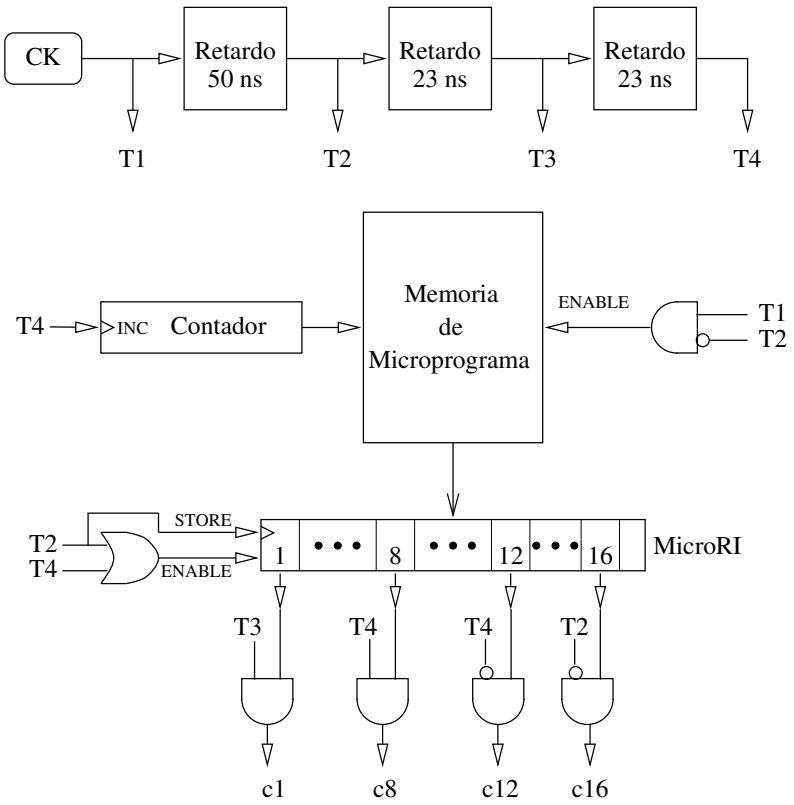


Figura 4.13: Esquema parcial de las conexiones de temporización

mador; en este caso, se conectará al conmutador c16 el AND de la línea de control c16 y de $\overline{T2}$, de forma que el desplazamiento esté controlado por el flanco de bajada de T2 y, por tanto, ocurrirá (si procede) unos 50 ns después de la habilitación del conmutador c8.

Finalmente, los conmutadores c9 a c12 deben habilitarse para terminar el ciclo (fase 7); en consecuencia, para cada una de sus respectivas señales de control deberá realizarse el AND con $\overline{T4}$, de forma que los registros I, A, B y/o D almacenarán la información desde el bus X en el flanco de bajada de la señal T4. La figura 4.14 muestra el diagrama de tiempos de las distintas fases durante un ciclo máquina del procesador.

En dicha figura, las líneas discontinuas indican que el punto de control asociado se activa sólo si el bit correspondiente vale 1.

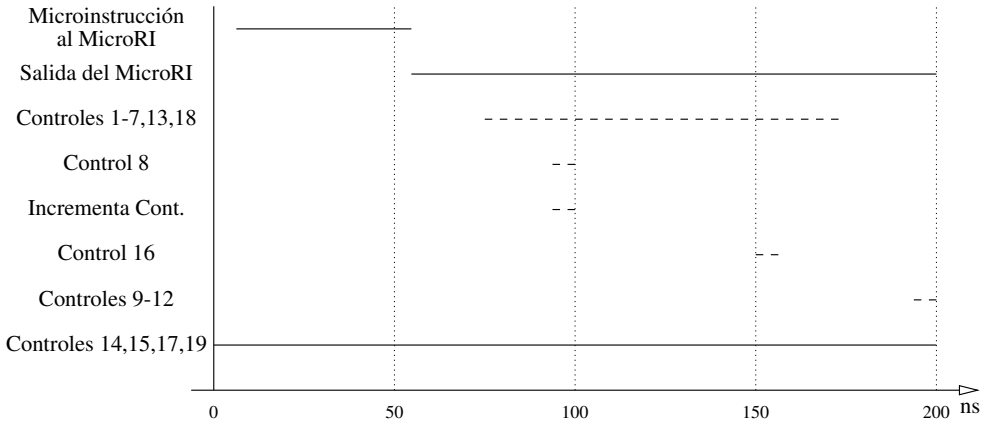


Figura 4.14: Diagrama de tiempos del ciclo máquina

4.3.2. Codificación de las microinstrucciones

Hasta este punto, la filosofía seguida en el diseño de la microinstrucción ha sido una filosofía que llamaremos **horizontal** o sin codificación, en la que a cada bit corresponde una señal de control. Sin embargo, los 19 bits empleados dan lugar a 524.288 combinaciones diferentes, cantidad mucho mayor que el número máximo de microinstrucciones distintas que se emplearán previsiblemente. Esto significa que la palabra de control contiene una redundancia considerable. Si el tamaño de la memoria de control es un factor importante en el coste final del diseño, será necesario reducir el tamaño de la microinstrucción introduciendo codificación, a costa de reducir la velocidad de ejecución.

La metodología básica consiste en localizar en la microinstrucción grupos de bits de control que sean mutuamente exclusivos o redundantes y codificarlos. El ejemplo más sencillo son los bits 1 al 4, que controlan cuál de los registros (I, A, B o cero) depositará su contenido en la entrada izquierda del sumador, L. Dado que sólo uno de estos cuatro registros puede ser seleccionado simultáneamente, es posible codificarlo usando tan sólo dos bits, de la siguiente manera:

Código	Registro
00	ceros
01	I
10	A
11	B

Para que esta representación funcione, es necesario incluir una lógica de decodificación entre el registro de microinstrucción y los conmutadores del *datapath*. Así, por ejemplo, si los bits empleados para codificar el registro seleccionado son el 1 y el 2, la señal de habilitación del conmutador c1 deberá cumplir la ecuación:

$$c1 = \overline{b_1} \ \ AND \ \ b_2 \ \ AND \ \ T_3$$

En la figura 4.15 se representa la lógica necesaria para la generación de esta señal. Una lógica de decodificación análoga deberá ser incluida para los conmutadores c2 a c4.

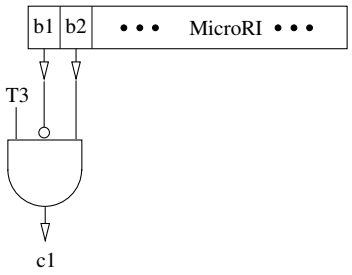


Figura 4.15: Decodificador de microfunción

Como se observa, al codificar la información contenida en los bits b1 y b2 del registro de microinstrucción, ahora ninguno de ellos tiene sentido por separado, sino que significan algo sólo cuando son analizados juntos. A este conjunto de uno o más bits en una microinstrucción que controlan un fragmento del *datapath* y que sólo tienen sentido al ser analizados juntos, es lo que habíamos denominado **microfunción** o **microorden**.

Examinando el *datapath*, es posible reducir aún más el número de bits de control. Así, los conmutadores c5 a c7, que controlan el registro origen en la entrada derecha del sumador, pueden codificarse con dos bits, quedando una combinación sin usar. Por otra parte, los conmutadores c9 a c12 controlan el destino del contenido del bus X. Teniendo en cuenta que no es necesario almacenar este contenido en más de un registro distinto y que es preciso incluir un caso en que el bus X no sea almacenado en ningún registro, deberán usarse tres bits, en los que se puede usar la siguiente codificación de destino:

Código	Destino
000	no operación
001	I
010	A
011	B
100	D (hace c14 = 0)
101-111	no usados

La inclusión de un código que no vuelca el contenido del bus X en ningún registro permite eliminar la necesidad de generar la señal de control para el conmutador c8, asociado al registro de desplazamiento. En efecto, cuando una microinstrucción usa el sumador esta señal debe ser 1; sin embargo, cuando la microinstrucción no usa el sumador, puede también dejarse esta señal a 1 y elegir el código 000 como destino del bus X. En consecuencia, se elimina el bit de control del conmutador c8, que estará controlado directamente por la señal de reloj T4.

Existe también redundancia entre los bits 12 a 14 (control de la entrada y salida del registro D) y los bits 18 y 19 (acceso a memoria y lectura/escritura, respectivamente). En efecto, en una operación de lectura de memoria (c18 y c19 a 1), debe habilitarse el conmutador c12, deshabilitarse el conmutador c13 y ponerse a 1 la línea c14 de selección en el multiplexor D-Mpx. Por el contrario, en una operación de escritura a memoria (bit 18 a 1, bit 19 a 0) el conmutador c12 debe permanecer deshabilitado, mientras el c13 deberá habilitarse. Teniendo en cuenta que el control del registro D en operaciones que no involucren a la memoria es efectuado por la microorden destino del bus X, resulta evidente que los bits 12 a 14 son superfluos, pues el estado de las señales de control c12 a c14 puede decodificarse de los bits 18 y 19 y de la microorden del bus X.

Como resumen de las simplificaciones sugeridas, en la tabla 4.5 se muestra la nueva definición de las microordenes, en la que se han reducido las necesidades de almacenamiento de control en un 37 %; adicionalmente, se ha reducido de forma significativa las posibilidades de codificar microinstrucciones erróneas o sin significado. El precio a pagar por ello es un aumento de la lógica de control necesaria y una pérdida de velocidad y posibilidad de funcionamiento concurrente. A esta filosofía de representación de microinstrucciones la hemos llamamos **vertical** o con codificación.

4.3.3. Expansión funcional del "datapath": ALU

La funcionalidad del *datapath* empleado hasta ahora como ejemplo es muy limitada debido al hecho de que la única operación que puede realizar es la

Microorden	Bits	Caso	Función
Entrada izquierda del Sumador	1-2	00	0
		01	I
		10	A
		11	B
Entrada derecha del Sumador	3-4	00	0
		01	B
		10	D
		11	No usada
Carry de entrada al Sumador	5	0	Carry=0
		1	Carry=1
Desplazamiento	6-7	00	No desplazamiento
		01	Desplazamiento Izq.
		10	Desplazamiento Der.
		11	No usada
Destino del BUS X	8-10	000	Ninguno
		001	I
		010	A
		011	B
		100	D
		101-111	No usada
Operación de Memoria	11-12	00	No operación
		01	Leer a D
		10	Escribir de D
		11	No usada

Tabla 4.5: Definición de microinstrucción codificada

suma. Este defecto es fácilmente solventable sustituyendo el sumador binario completo por una ALU capaz de realizar las siguientes funciones: $L + R + C$; $L + \overline{R} + C$; $L\ OR\ R$ y $L\ AND\ R$.

Con cuatro posibles operaciones para ejecutar, se necesitarán dos bits para su selección, c20 y c21, por lo que deberá incrementarse en dos bits el tamaño de la microinstrucción. En la figura 4.16 vemos que esta ALU también proporciona dos *flags* de estado, C (carry) y Z (cero), que son almacenados en un registro de **FLAGS**. Un bit de control adicional, c22, señalará la carga de dicho registro.

4.3.4. Capacidad de salto y constantes en la microinstrucción

El campo "dirección de salto" en las microinstrucciones sólo es útil en aquellas que supongan un salto en el microprograma, siendo un conjunto de bits desaprovechados en el resto. No obstante, podemos ampliar la funcionalidad

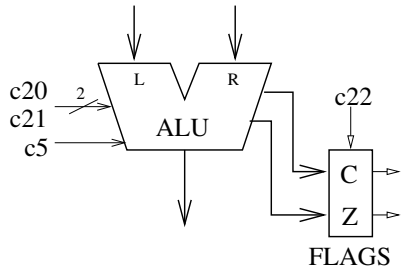


Figura 4.16: Ampliación de la ALU

de las microinstrucciones, permitiendo que, en las instrucciones de no salto, el contenido de este campo pueda ser usado como una constante (valor) en la microinstrucción. Para ello, abrimos un camino de datos que permita cargar en la ALU el contenido del campo "dirección de salto" del registro MicroRI. Este camino estará controlado con un nuevo punto de control, c23, que se muestra en la figura 4.17. A este campo le llamaremos desde ahora "dirección/constante".

Respecto a los esquemas de temporización necesarios para el buen funcionamiento de estas unidades de procesamiento y control mejoradas, es necesario realizar un cambio importante relativo al momento en que se debe cargar el MicroPC. Es decir, si queremos que en una misma microinstrucción se pueda realizar una operación aritmética y un salto condicional al resultado de ésta, tendremos que permitir la carga o el incremento del MicroPC una vez haya concluido la operación en la ALU y se haya actualizado convenientemente el registro de FLAGS. Por ejemplo, en la fase 5 de la página 92, en que se habilita c8 para cargar el registro de desplazamiento con el resultado de la ALU, también podemos activar c22 para actualizar el registro de FLAGS si es necesario, es decir, con el flanco de subida de T4. De esta forma, el incremento o carga del MicroPC, condicionada al valor de los flags, se puede realizar con el flanco de bajada de T3 o T4.

4.3.5. Microprogramación del conjunto de instrucciones

Para que el conjunto datapath y sección de control pueda ser usado como un procesador de propósito general, añadiremos un sistema de proyección a continuación de IR y un contador de programa PC. De esta forma, el procesador final, incluida la sección de control, sería el mostrado en la figura 4.17, en el que se muestran todos los puntos de control incluídos los adicionales (carga de constantes,...). Puesto que hemos empleado codificación en la microinstruc-

ción, es necesario usar un decodificador a la salida de MicroRI. El significado de los bits que componen la microinstrucción se muestra en la tabla 4.6.

Microorden	Bits	Caso	Función
Entrada izquierda del Sumador	1-2	00 01 10 11	0 I A B
Entrada derecha del Sumador	3-5	000 001 010 011 100 101-111	0 B D PC Constante (bits 21-30) No usado
Carry de entrada al Sumador	6	0 1	Carry=0 Carry=1
Desplazamiento	7-8	00 01 10 11	No desplazamiento Desplazamiento Izq. Desplazamiento Der. No usada
Destino del BUS X	9-11	000 001 010 011 100 101 110-111	Ninguno I A B D PC No usada
Operación de Memoria	12-14	000 001 010 011 100 101-111	No operación (A) → D D → (A) (PC) → RI (PC) → D No usada
Función de la ALU	15-16	00 01 10 11	Suma $(L + R + C)$ Resta $(L + \overline{R} + C)$ OR $(L\ OR\ R)$ AND $(L\ AND\ R)$
Carga del Registro de FLAGS	17	0 1	No modifica el registro Actualiza el registro
Control de MUX2	18	0 1	Pasa Dirección de MicroRI Pasa Dirección de ROM
Control de Salto	19-20	00 01 10 11	μPC=μPC+1 Salto Condicional si C Salto Condicional si Z μ PC = Dirección
Dir. Salto/Constante	21-30		

Tabla 4.6: Definición final de microinstrucción

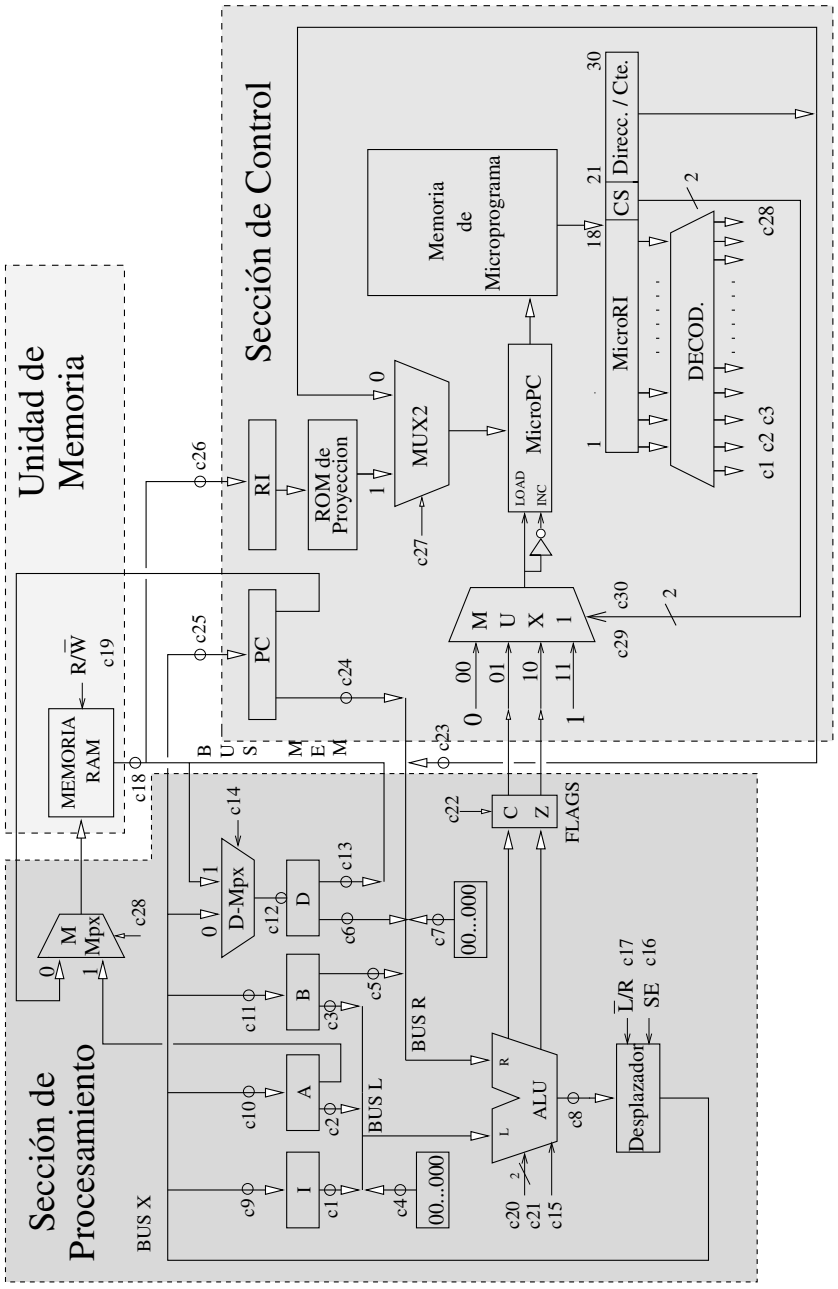


Figura 4.17: Procesador hipotético completo. Secciones de control y procesamiento con codificación horizontal

Aclararemos con un ejemplo el problema de escribir una microrrutina asociada a una instrucción máquina dada para este procesador. Consideremos la instrucción ejemplo:

AND 25(A),B

Esta instrucción debe almacenar en el registro B el resultado de hacer el AND lógico entre el contenido de este mismo registro y el contenido de la posición de memoria apuntada por el registro A más 25. Es decir:

$$(25 + A)\ \ AND\ \ B \rightarrow B$$

Dado que el C.O. ocupa la palabra completa de memoria, el desplazamiento, 25, del direccionamiento relativo, se almacena en la siguiente palabra. Por tanto, las microinstrucciones que se deben ejecutar son las siguientes:

1. $RI \leftarrow (PC)$: Cargar el contenido de memoria apuntado por el PC en RI, es decir, **búsqueda de instrucción**.
2. $c27 \leftarrow 1$; $CS = 11$; $PC \leftarrow PC + 1$: **Decodificación** (carga del MicroPC con la dirección que viene de la ROM de Proyección) e incremento del PC.
3. $D \leftarrow (PC)$; $PC \leftarrow PC + 1$: Extraemos el desplazamiento de la memoria (el "25") e incrementamos el PC para que apunte a la siguiente instrucción.
4. $A \leftarrow A+D$: Completamos el **cálculo de la dirección efectiva** sumando al registro A el desplazamiento previamente almacenado en D.
5. $D \leftarrow (A)$: **Búsqueda del operando** en memoria a partir de su dirección efectiva almacenada en A.
6. $B \leftarrow (B\ \ AND\ \ D)$; $CS = 11$; $Dir = 0$: **Ejecución**, realizando el AND lógico, y salto final a la posición 0, donde se encuentra la rutina de fetching.

Siguiendo la codificación de las microordenes presentada en la tabla 4.6 y suponiendo que el núcleo de la microrrutina se almacena a partir de la posición 12 de la memoria de microprograma, tendremos que el contenido parcial de esta memoria es el siguiente:

```
 Dirección                          Contenido
 ^^^^^^^^^                          ^^^^^^^^^
00000 00000          00 000 0 00 000 011 00 0 0 00 0000000000
00000 00001          00 011 1 00 101 000 00 0 1 11 0000000000
     .
     .
00000 01100          00 011 1 00 101 100 00 0 0 00 0000000000
00000 01101          10 010 0 00 010 000 00 0 0 00 0000000000
00000 01110          00 000 0 00 000 001 00 0 0 00 0000000000
00000 01111          11 010 0 00 011 000 11 1 0 11 0000000000
```

La ROM de proyección debe contener la traducción entre códigos de operación y dirección de microrutina asociada a la instrucción.

Como vemos en este ejemplo, el código de operación de cada instrucción contiene en este caso información, no sólo de la operación a realizar y del modo de direccionamiento, sino también de los operandos. Esto implica que, según nuestro ejemplo, la instrucción **AND 10(A), I** debe tener una microrrutina distinta a la descrita anteriormente, con el consiguiente consumo de Memoria de Microprograma. En diseños más elaborados, el C.O. no ocupará todo el RI, dejando espacio para especificar los registros que se vayan a acceder, los cuales serán seleccionados directamente desde el propio RI.

4.4. DISEÑO DE UNA UNIDAD DE CONTROL CABLEADA

El diseño de unidades de control cableadas implica un compromiso entre la cantidad de hardware usado, su velocidad de operación y el coste de su proceso de diseño. Los métodos de diseño utilizados en la práctica son, a menudo, de naturaleza heurística y ad hoc (no generales, sino orientados al problema concreto a resolver). Además, estos métodos de diseño no pueden formalizarse fácilmente, debido al gran número de señales de control necesarias en una unidad de control y a su dependencia con el conjunto de instrucciones particular que se implementa.

Para ilustrar el problema, consideramos dos aproximaciones simplificadas y sistemáticas para el diseño de unidades de control cableadas. Estos métodos son representativos de los que se usan en la práctica, pero son adecuados únicamente para unidades de control sencillas, tales como las que se pueden encontrar en controladores no programables o en procesadores RISC. Estas dos aproximaciones se denominan respectivamente **método de los elementos de retardo** y **método de los contadores de secuencia**.

Vamos a explicar estos métodos sobre un ejemplo, diseñando una unidad

de control para la sección de procesamiento que se presentó anteriormente en la figura 4.3. Asociada a esta sección se detallaron los puntos de control en la tabla 4.1 .

Desde el punto de vista del diseño consideraremos que los señales de control del procesador son activas a nivel alto. No obstante aquellas que actúen por flanco de subida o bajada sobre el commutador, (carga, desplazamiento, incremento de registros) tendrán que ser combinada con la señal de reloj.

Consideraremos un repertorio de instrucciones reducido, que se muestra en la tabla 4.7[8].

Mnemónico	Descripción
LOAD A	Transfiere el contenido de la posición A al acumulador: $(A) \rightarrow ACC$
STORE A	Transfiere el contenido del acumulador a la posición A: $ACC \rightarrow (A)$
ADD A	Suma al acumulador el contenido de la posición A: $ACC + (A) \rightarrow ACC$
JMPZ A	Salta a la dirección A si el resultado fue cero: $si\ Z : A \rightarrow PC$

Tabla 4.7: Repertorio de instrucciones

Vamos a aplicar los métodos antes citados para la construcción de este procesador simplificado. En la figura 4.18 se representa el diagrama de flujo del funcionamiento del procesador. `BEGIN` representa una señal de inicio. Como se puede observar la tarea del procesador se divide en dos fases básicas: búsqueda de instrucción y ejecución. Durante la búsqueda se accede a la posición de memoria apuntada por el PC y se transfiere ese contenido al registro de instrucción IR. El PC es incrementado apuntando ahora a la siguiente dirección de memoria. El contenido de IR se decodifica (identificación de a qué instrucción pertenece el código de operación) y según la operación a realizar se ejecuta una u otra cosa. En la fase de ejecución se busca el operando, se ejecuta la operación de ALU y/o se almacena datos en memoria según proceda.

4.4.1. Método de los elementos de retardo

En este método se parte del diagrama de flujo de control para crear el circuito encargado de producir las señales de control. Para ello lo que vamos a hacer es generar el conjunto de señales necesarias para cada operación elemental (transferencia de registros) que hemos escrito en el diagrama de flujo, en el orden adecuado. Cada etapa del diagrama de flujo lo denominaremos microoperación.

[8]Con (A) nos estamos refiriendo al contenido de la posición A de memoria

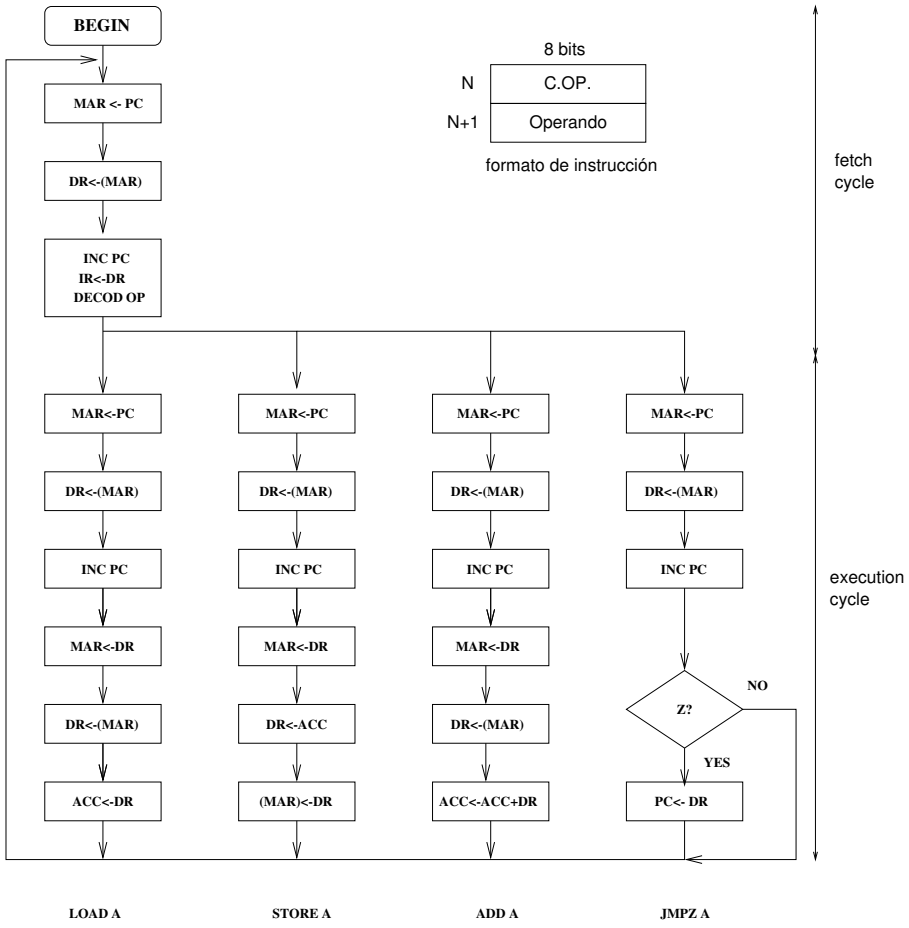


Figura 4.18: Diagrama de flujo a nivel RTL para el procesador de la fig. 4.3

Vamos a introducir un conjunto de señales auxiliares C_{ji} cuyo significado es: "C_{ji} se activa en un intervalo t_i si la señal de control C_j debe estar activa en ese intervalo".

Consideremos el problema de generar la siguiente secuencia de señales de control en los instantes t_1,t_2,...,t_n:

```
t1: Activar Cj1;
t2: Activar Cj2;
...
tn: Activar Cjn;
```

Aquí con C_{ji} representamos la activación del punto de control C_j en el

instante t_i. (¡Cuidado! no todas las señales se activan en todos los instantes; en cada instante se activarán aquel subconjunto de señales que intervenga en la transferencia de registros que queramos realizar). En cada ciclo o intervalo t_i se ejecuta una microoperación.

Podemos producir la cadencia de señales de control introduciendo elementos de retardo ("delay element") y propagando la señal de activación. Con este método la translación entre el diagrama de flujo y la sección de control es directa ya que el diagrama de flujo expresa la secuencia de activación de los puntos de control. El circuito diseñado tiene esencialmente la misma estructura que el diagrama de flujo. La forma de obtener el circuito a partir del diagrama de flujo nos la indica unas reglas simples, que se ven en la figura 4.19. A continuación se explican dichas reglas.

1. Cada secuencia de dos microoperaciones sucesivas requiere un elemento de retardo. Las señales que activan las líneas de control se toman directamente de las líneas de entrada y salida del circuito de retardo. Las señales que activan una misma línea de control C_i se unen con una puerta OR cuya salida es C_i. Esta línea puede conectarse al punto de control que activa (introduciendo la temporización adecuada si fuera preciso).
2. k líneas en el diagrama de flujo que se unen a una línea común se transforma en una puerta OR de k entradas, como muestra la figura 4.19
3. Un bloque de decisión (operación de salto condicional) o bifurcación se implementa con puertas AND, como aparece en la figura 4.19.

No olvidemos que las señales de control así generadas C_j pueden necesitar ser combinadas con la señal de reloj para generar los flancos adecuados cuando conectemos las señales con la sección de procesamiento.

Aplicando esta filosofía el resultado para el procesador que estamos construyendo es el que se muestra en la figura 4.20.

En este esquema hay que tener en cuenta:

- Como se puede observar el código de operación contenido en IR, es decodificado para producir 4 señales (LOAD, STORE, ADD, JMPZ) que nos sirven para escoger qué rama del diagrama de flujo seguir según la instrucción que represente el código de operación.
- Es necesario introducir una señal BEGIN (no aparece en el esquema) que ponga a cero (CLEAR) todos los elementos de retardo (y así todas las señales de control) salvo el primer elemento (señalado con '$*$') de retardo que se pondrá a uno (PRESET) cuando comience la ejecución. Esta señal así mismo debera poner el PC a un valor conocido apuntando a la primera instrucción a ejecutar (lo más sencillo sería iniciarlo a cero).

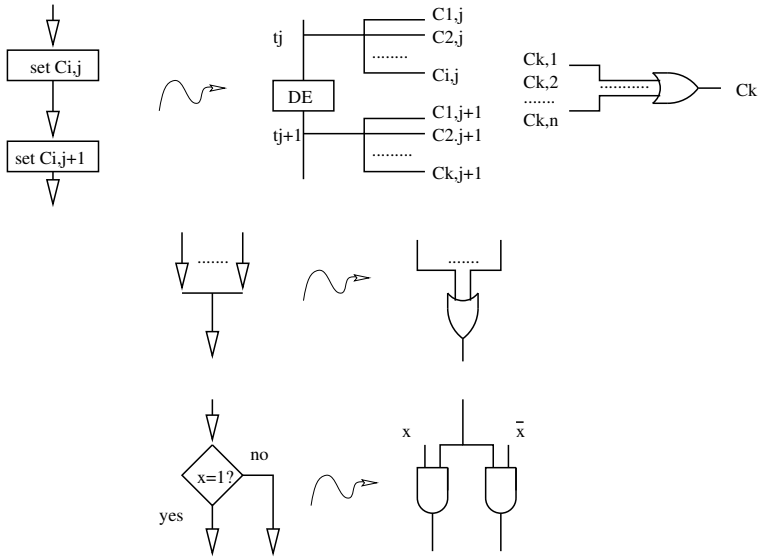


Figura 4.19: Transformación entre diagrama de flujo y circuito con elementos de retardo

- Existen señales como $C_{8,4}$, $C_{8,10}$, $C_{8,16}$, $C_{8,22}$ que no se pueden activar nunca a la vez ya que pertenecen a "ramas" excluyentes en el flujo de ejecución. Es por esta razón por lo que se ha usado una numeración para el segundo subíndice diferente, aunque las 4 señales, cuando se activan, lo harían en el intervalo ficticio t_4. (Es decir dentro del ciclo de ejecución de una instrucción $t_4,\ t_{10},\ t_{16},\ t_{22}$ representan el mismo intervalo temporal real).

Una vez que se posee el circuito que en cada instante genera los C_{ji} adecuadamente, hay que generar las señales de control C_j. Como el significado de C_{ji} es la activación de la señal C_j en el instante t_i, los C_j los determinamos con el OR de todos los C_{ji} que se activan para algún t_i. Las ecuaciones que resultan son las siguientes:

$$C_j = \sum_i C_{ji}$$

El elemento de retardo (DE), se construye simplemente con un biestable tipo D activo por flanco (por ejemplo de subida). Construido de esta manera la señal de reloj de los elementos de retardo debe ser aquella que sincronice también los puntos de control activos por flanco en la sección de datos. Así por

ejemplo la señal C_7 que controla la carga del registro DR debe ser combinada con la señal de reloj cuando es conectada en el punto de control $c7$: $c7 \equiv C_7 \cdot \overline{CK}$; o para el acceso a memoria $c3 \equiv C_3 + \overline{CK}$ ya que cuando se escribe el dato se almacena en el flanco de subida de c3.

4.4.2. Método del contador de secuencias

Consideremos el circuito de la figura 4.21, que consiste basicamente en un contador módulo k cuya salida esta conectada a un decodificador $\frac{1}{k}$. Si la entrada ENABLE del contador se conecta a un reloj fuente, el contador ira pasando continuamente por sus k estados. El decodificador genera k señales pulso S_j en su salida y, además, pulsos consecutivos están separados por un periodo de reloj, como muestra la figura. Las señales S_j dividen el tiempo requerido para que un contador recorra su ciclo completo en k partes iguales. A estas señales S_j se les llama señales de fase. Se necesita un flip-flop y dos líneas de entrada para poner en marcha o parar el contador. Un pulso en la línea BEGIN hace que el contador comience a recorrer sus estados cíclicamente, conectando la línea ENABLE del contador al reloj (CK) fuente. Un pulso en la línea END desconecta el reloj y resetea el contador. Supondremos que la señal de "reset" del contador es síncrona con la entrada de reloj. El circuito de la figura 4.21 se llama contador de secuencia y lo representaremos tal como se indica en dicha figura.

La utilidad de los contadores de control de este tipo radica en el hecho de que muchos circuitos digitales se diseñan para realizar un número relativamente pequeño de acciones repetidamente.

Cada paso a través del lazo constituye un ciclo de microoperación. Observando el diagrama de flujo de la figura 4.18 vemos que desde que una instrucción comienza su ejecución, en el caso peor se consumen 9 etapas (longitud del camino mas largo). Si suponemos que cada etapa puede realizarse en un periodo de reloj elegido, entonces podremos construir una unidad de control para esta CPU, partiendo de un simple contador de secuencia módulo 9. Cada señal S_i activara las líneas de control que procedan, en la etapa i-ésima.

El diseño del circuito que genera las señales de control se reduce a un combinacional que activará cada control C_j en función del código de operación, del contenido del registro de estados y de la etapa (S_i) en la que se encuentre la ejecución. El combinacional, que representamos en la figura 4.22, debe implementar las ecuaciones lógicas que se muestran:

$$C_0 = ADD \cdot S_9$$
$$C_1 = 0$$

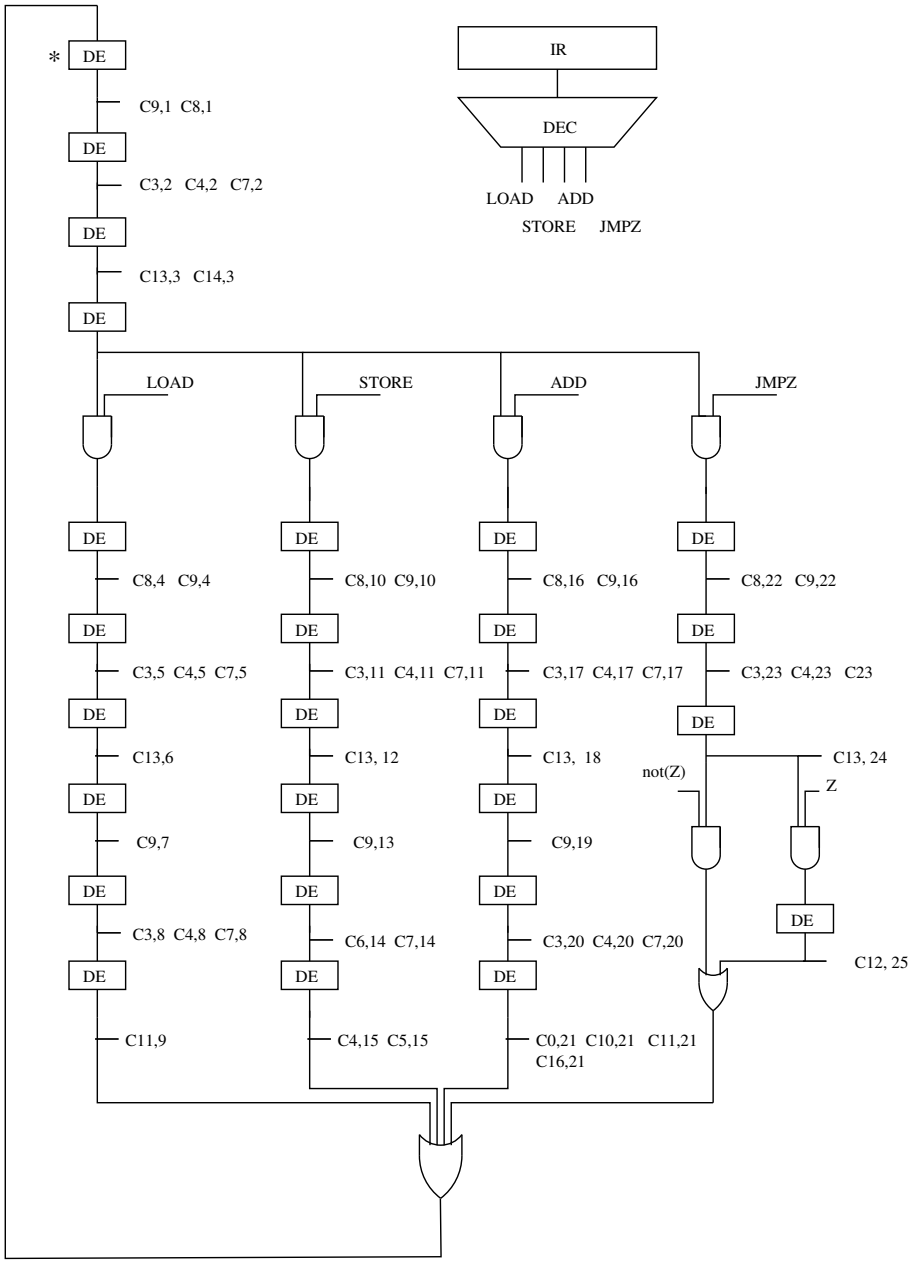


Figura 4.20: Señales de control con elementos de retardo

$$C_2 = 0$$
$$C_3 = S_2 + LOAD \cdot S_5 + STORE \cdot S_5 + ADD \cdot S_5 + JMPZ \cdot S_5 + LOAD \cdot S_8 + ADD \cdot S_8$$
$$C_4 = S_2 + LOAD \cdot S_5 + STORE \cdot S_5 + ADD \cdot S_5 + JMPZ \cdot S_5 + LOAD \cdot S_8 + ADD \cdot S_8$$
$$C_5 = STORE \cdot S_9$$
$$C_6 = STORE \cdot S_8$$
$$C_7 = S_2 + LOAD \cdot S_5 + STORE \cdot S_5 + ADD \cdot S_5 + JMPZ \cdot S_5 + LOAD \cdot S_8 + STORE \cdot S_8 + ADD \cdot S_8$$
$$C_8 = LOAD \cdot S_5 + STORE \cdot S_5 + ADD \cdot S_5 + JMPZ \cdot S_5 + S_1$$
$$C_9 = S_1 + LOAD \cdot S_7 + STORE \cdot S_7 + ADD \cdot S_7 + S_4$$
$$C_{10} = ADD \cdot S_9$$
$$C_{11} = LOAD \cdot S_9 + ADD \cdot S_9$$
$$C_{12} = JMPZ \cdot S_7$$
$$C_{13} = S_3 + S_6$$
$$C_{14} = S_3$$
$$C_{15} = 0$$
$$C_{16} = ADD \cdot S_9$$
$$RESET = JMPZ \cdot S_8 \cdot Z + JMPZ \cdot S_7 \cdot \overline{Z}$$

Sobre estas ecuaciones podemos comentar:

- Su obtención es inmediata a partir del significado de las señales. Así por ejemplo C_3 nos indica que se activa siempre en la segunda etapa, y en la etapa 5 cuando la instrucción es un LOAD, STORE, ADD o JMPZ, y en la etapa 8 cuando se trata de un LOAD o un ADD.
- Aunque se han expresado de forma que se entienda como se obtienen, las expresiones son susceptible de simplificación, más aún teniendo en cuenta que expresiones mutuamente exclusivas siempre ocurren, por ejemplo $LOAD + STORE + ADD + JMPZ = 1$.
- Hay señales que nunca se activan en este conjunto de instrucciones (C_1, C_2, C_{15}).
- No todas las instrucciones consumen la 9 etapas. La instrucción JMPZ consume menos. Por eso hay que introducir la señal RESET (no olvidar que es síncrona con el reloj). Esta señal se activa sólo para JMPZ, cuya ejecución finaliza en la etapa 8 si Z está activo (salto) o en la etapa 7 si Z está inactivo (no salto).

Como se indicó para el método anterior, las señales de control deben ser combinadas con el reloj en aquellos casos en el que el conmutador de control funcione por flanco.

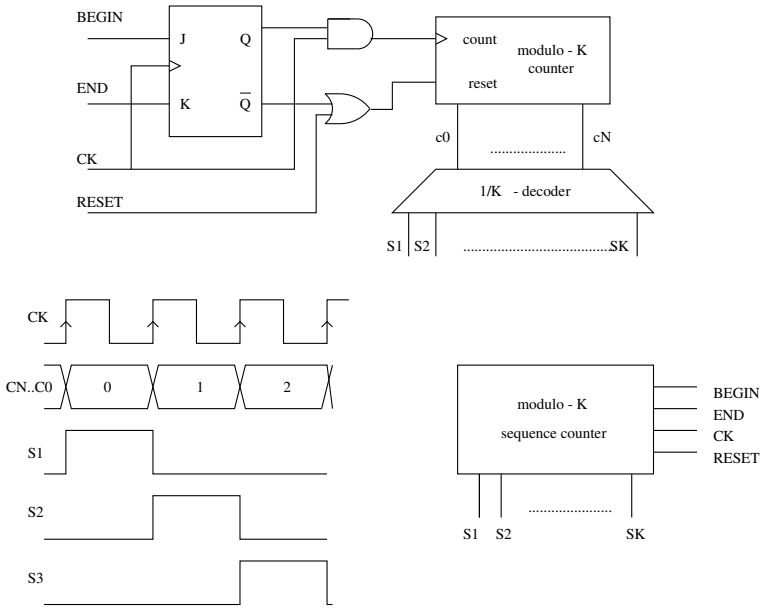


Figura 4.21: Contador de secuencias módulo k

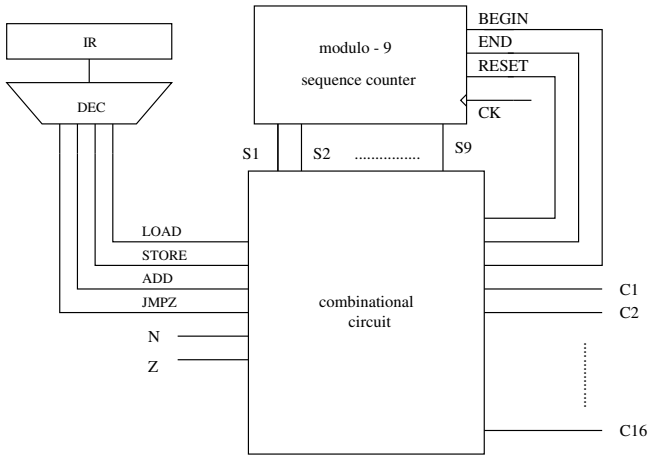


Figura 4.22: Unidad de control utilizando un contador de secuencias

SINOPSIS

La técnica de control microprogramado permite el diseño de una unidad de control de forma sistemática y estructurada. En resumen, hemos visto que

básicamente necesitamos incluir dentro de la sección de control un contador de microprograma, **MicroPC**, un registro de microinstrucción, **MicroRI** y una **Memoria de Microprograma**, donde almacenamos cada una de las microrrutinas asociadas a cada instrucción máquina. Además, hemos introducido la metodología de diseño de unidades de control cableada, usadas en procesadores RISC.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

RELACIÓN DE PROBLEMAS

1. Para la máquina descrita por la figura 4.17 y la tabla 4.6 de la sección 4.3.5, diseñar la microsubrutina asociada a la instrucción máquina **BEQ** *dir*, que carga en el PC el valor *dir* si el flag Z es igual a 1. La instrucción **BEQ** *dir* ocupa dos palabras de memoria (una para el código de operación y otra para la dirección de salto).
2. Diseñar la microsubrutina asociada a la instrucción máquina **MAX A,B** que coloca en el registro B el valor *máximo(A,B)*. El contenido de los registros A y B será siempre positivo y representado en C2.
3. Ídem para la instrucción **EXG A,B** que intercambia los valores de los registros A y B.
4. Sean las secciones de procesamiento y de control microprogramado de la figura 4.23. Sus elementos más destacados son:
   - Memoria RAM externa común para datos e instrucciones asociada a dos registros: uno de direcciones, MAR y otro de datos, DR. En una operación de lectura se cargará DR con el contenido de la memoria apuntada por MAR. En escritura, el contenido de DR se almacenará en la posición de memoria indicada por MAR.
   - Tres registros de propósito general A, B y TMP, más un registro a cero ("0").
   - Registro de Instrucción (RI): Almacena una instrucción con un código de operación de 4 bits.
   - Contador de Programa (PC): Puede ser cargado desde el BUS, volcado al BUS e incrementado.
   - Unidad Aritmético-Lógica capaz de realizar las 4 operaciones básicas de suma, resta, producto y división. Dispone de dos flags de estado: Signo (N) y Cero (Z).

   Suponer resueltos los posibles problemas de temporización y el siguiente tiempo de respuesta para el hardware:
   - 1 ciclo de reloj para completar la siguiente cadena de operaciones: "Volcado de alguno de los registros a la ALU"; "Operación ALU" "Escritura en alguno de los registros". La carga en el MicroPC se realiza al mismo tiempo que la escritura en registros, es decir, al final del ciclo de reloj.
   - 1 ciclo de reloj para la lectura o escritura del dato almacenado en DR en la posición de memoria dada por MAR.
   - 1 ciclo para cualquier transferencia entre los registros DR, RI, PC

y MAR a través del BUS.

Se pide:

a) Diseñar la microrrutina de $fetch$ y decodificación. Describirla en *pseudocódigo* y detallar el contenido de la memoria de microprograma en binario a partir de la dirección 0.

b) Diseñar las microrrutinas asociadas a cada una de las instrucciones de la tabla 4.10. Descríbelas en *pseudocódigo* y detalla el contenido de la memoria de microprograma en binario, colocando cada microrrutina a partir de la posición de micromemoria que le corresponda. Utiliza la tabla 4.9 para escribir las microsubrutinas, utilizando el campo **Comentario** para poner en pseudocódigo el significado de cada instrucción. Suponer que el ciclo de $fetch$ no modifica los flag de estado.

c) Detallar el número de palabras y el contenido de la ROM de Proyección teniendo en cuenta el código de operación de cada instrucción de la tabla 4.10.

SEÑAL	Significado
c1, c2, c3	Habilitación de entrada a registros A, B y TMP.
c4	Habilitación de entrada a registro RI.
c5, c6, c7, c8	Habilitación de salida de A, B, TMP y "0" a la ALU.
c9	Control del Multiplexor-1.
c10	Vuelca al BUS el dato de salida de la ALU.
c11, c12	Selecciona la operación al realizar en la ALU: 00=Suma. 01=Resta. 10=Producto. 11=División.
c13	Incrementa el PC.
c14	Carga en PC el dato que haya en el BUS.
c15	Vuelca al BUS el dato almacenado en PC.
c16	Debe estar a 1 cada vez que se utilice la memoria (Chip Select).
c17	1 = Lectura de memoria a DR. 0 = Escritura en memoria de DR.
c18	Carga en MAR el dato que haya en el BUS.
c19	Carga en DR el dato que haya en el BUS.
c20	Vuelca al BUS el dato almacenado en DR.

Tabla 4.8: Puntos de control especificados (Prob. 4)

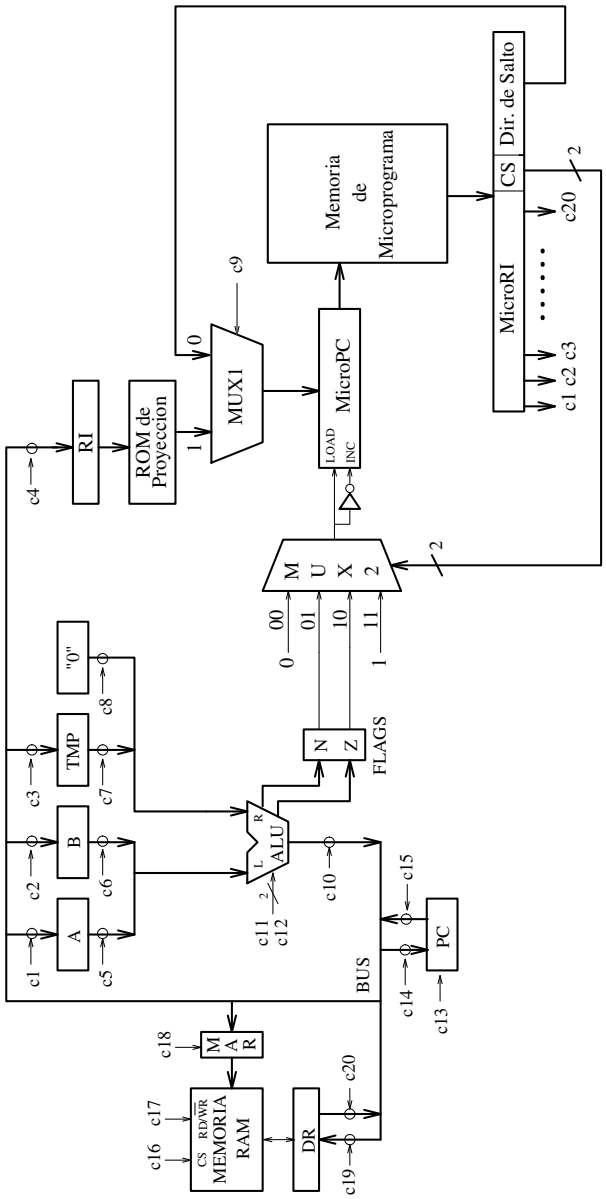


Figura 4.23: Unidades de Proceso y de Control (Prob. 4)

	Registros								M	ALU			PC			Mem.		MAR	DR			Salto	
Comentario	Dir.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20	CS	Dir. Salto
	0																						
	1																						
	2																						
	3																						
	4																						
	5																						
	6																						
	7																						
	8																						
	9																						
	10																						
	11																						
	12																						
	13																						
	14																						
	15																						
	16																						
	17																						

Tabla 4.9: Memoria de microprograma (Prob. 4)

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Instrucción	OpCode	Tarea a realizar
MOVE A, TMP	0111	TMP = A
MOVE (A), B	1001	B = MEM[A]
BNE A	0011	IF (flag_Z = 0) THEN PC = A
MUL (A), B	0100	B = B * MEM[A]

Tabla 4.10: Conjunto de instrucciones (Prob. 4)

5. Sean las secciones de procesamiento y de control de la figura 4.24. La sección de procesamiento está diseñada para permitir eficientemente la multiplicación de dos números enteros positivos de 8 bits ($A \times B$, considerando que para que A y B sean positivos deben tener su bit más significativo a 0).

   Inicialmente el multiplicando, A, se almacena en el registro Y, el multiplicador, B, en el registro X, y el registro AC debe ser puesto a 0. Durante 8 iteraciones haremos lo siguiente: Si el bit menos significativo de X está a 1 ($X(0) = 1$), acumularemos en el registro AC su contenido más el de Y y desplazaremos un bit a la derecha el par de registros $AC - X$. Si, por el contrario, X(0)=0, sencillamente desplazaremos sin sumar previamente. En el desplazamiento a la derecha, el bit menos significativo de AC pasará a ser el más significativo del registro X. Después de 8 iteraciones, el resultado de la multiplicación (de 16 bits) queda en el par de registros $AC - X$. Un contador de cuatro bits, que puede ser inicializado a 0 y que puede ser incrementado, nos permite saber cuando se han producido las 8 iteraciones.

   En la tabla 4.11 se describe la función de cada uno de los 9 puntos de control implicados en este diseño. El formato de cada microinstrucción no codifica puntos de control en microordenes, sino que asigna un bit del registro de microinstrucción por cada punto de control. Además añade un campo de condición de salto, CS, para controlar el Multiplexor-2 y un campo de dirección de salto. El punto de control $c9$ permite la carga del contador de microprograma con la dirección de salto (si vale 0), o con la dirección que proporciona la ROM de Proyección (si vale 1). Teniendo en cuenta las siguientes suposiciones:

   - Los problemas de sincronización y temporales, relativos a los puntos de control, se consideran resueltos. Si la señal $c7$ se activa en una microinstrucción de salto, el incremento del contador se produce antes que la carga del contador de microprograma.

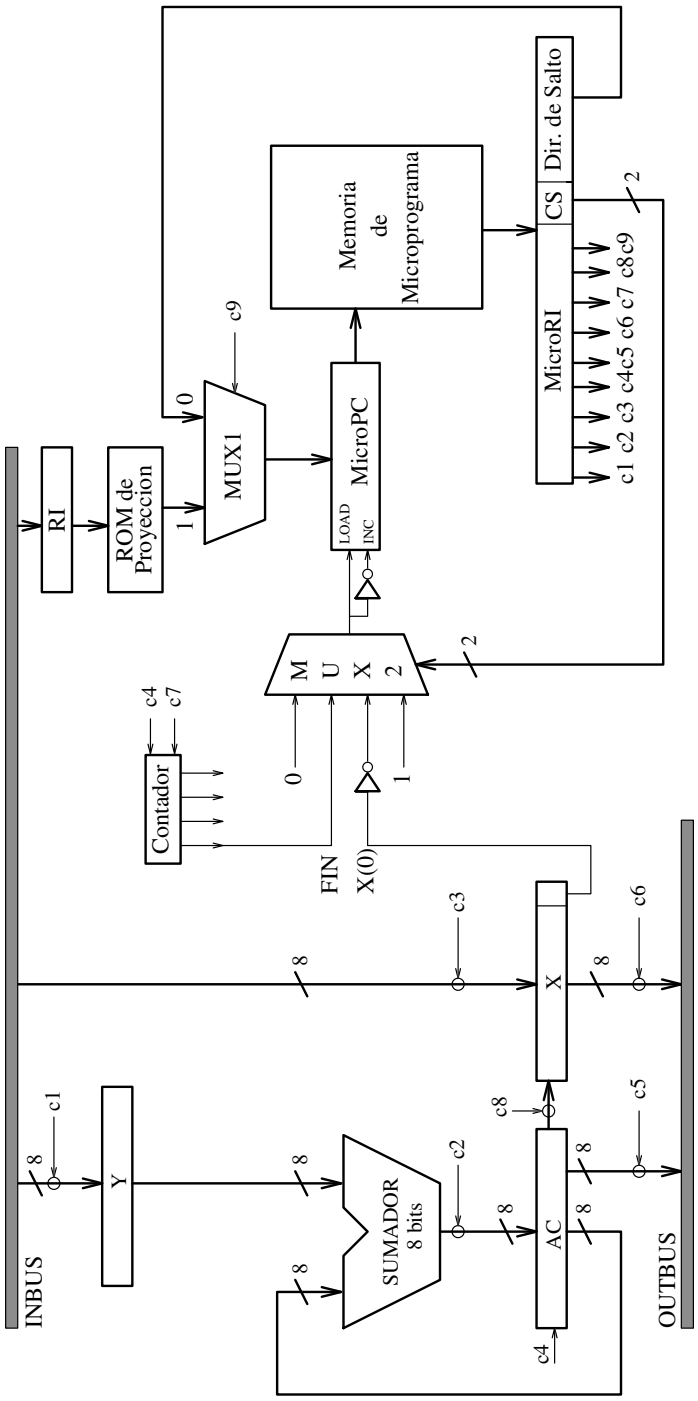


Figura 4.24: Unidades de Proceso y de Control del multiplicador

- La instrucción máquina MUL, de 0 direcciones, toma los operandos implícitamente de los registros X e Y y deja el resultado en los registros AC y X. Evidentemente, deben existir instrucciones máquina para la carga de los registros X e Y y para la descarga de los registros AC y X, pero consideraremos ese problema de microprogramación ya resuelto.
- El código de operación de la instrucción **MUL** es 24. En la posición 24 de la ROM de Proyección está almacenada la dirección 100.

Se pide:

- Escribir la microrutina asociada a la instrucción **MUL** y detallar el contenido binario de la memoria de microprograma, indicando la dirección inicial de la microrutina según las suposiciones expuestas anteriormente. Suponer el problema de la búsqueda y decodificación de la instrucción **MUL** ya resuelto.

Señal	Nemotécnico	Significado
c1	loadY	Carga en Y el valor de INBUS.
c2	loadAC	Carga en AC el resultado del sumador.
c3	loadX	Carga en X el valor de INBUS.
c4	clearACyC	Pone a cero el registro AC y el Contador.
c5	busAC	Saca a OUTBUS el valor de AC.
c6	busX	Saca a OUTBUS el valor de X.
c7	IncC	Incrementa el Contador.
c8	shift	Desplaza a derecha AC y X.
c9	C-Mux1	Control del Multiplexor-1.

Tabla 4.11: Puntos de control del multiplicador (Prob. 5)

6. Sean las secciones de procesamiento y de control microprogramado de la figura 4.25. Sus elementos más destacados son:
   - Memoria RAM externa, común para datos e instrucciones, asociada a un registro de datos, DR. En una operación de lectura se cargará DR con el contenido de la posición de memoria apuntada por el bus Y. En escritura, el contenido de DR se almacenará en la posición de memoria indicada por Y.
   - Un banco de cuatro registros entre los que se encuentra el Contador de Programa (PC). Las señales de control $c1$ y $c2$ seleccionan uno de los cuatro registros para volcar su contenido al bus X (si $c5$ está a 1), o para cargarse con el resultado de la ALU (si $c6$ está a

1). Los puntos de control $c3, c4$ seleccionan el registro que volcará su contenido al bus Y (independientemente de $c1$, $c2$, $c5$ y $c6$).

- Acumulador (AC): Puede ser cargado desde el BUS y proporciona uno de los operandos de la ALU.
- Registro de Instrucción (RI): Almacena una instrucción con un código de operación de 4 bits.
- Unidad Aritmético-Lógica capaz de realizar las 4 operaciones básicas de suma, resta, producto y división. Dispone de dos flags de estado: Signo (N) y Cero (Z).

Suponer resueltos los posibles problemas de temporización y el siguiente tiempo de respuesta para el hardware:

- 1 ciclo de reloj para completar la siguiente cadena de operaciones: "Volcado de alguno de los registros al BUS"; "Operación ALU del dato volcado con el que hay en AC" y "Escritura en alguno de los registros". Evidentemente, si $c5$ y $c6$ se activan en una misma microinstrucción, el registro que vuelca su contenido al bus X al comienzo del ciclo, es el mismo que se modifica al final de dicho ciclo máquina. La carga en el MicroPC se realiza al mismo tiempo que la escritura en registros, es decir, al final del ciclo de reloj.
- 1 ciclo de reloj para la lectura o escritura del dato almacenado en DR en la posición de memoria dada por el bus Y.
- 1 ciclo para cualquier transferencia entre los registros DR, RI, AC y alguno del Banco de Registros.
- El registro de estado también se carga al final del ciclo máquina si la señal $c15$ está activa. Sólo las instrucciones aritméticas o lógicas deben modificar el registro de estado.

Se pide:

a) Diseñar la microrrutina de $fetch$ y decodificación. Descríbela en *pseudocódigo* y detalla el contenido de la memoria de microprograma en binario a partir de la dirección 0. Utiliza la tabla adjunta para detallar las microinstrucciones, utilizando la columna **Comentario** para indicar en pseudocódigo el significado de cada microinstrucción.

b) Diseñar las microrrutinas asociadas a cada una de las instrucciones de la tabla 4.12. Descríbelas en *pseudocódigo* y detalla el contenido de la memoria de microprograma en binario, colocando cada microrrutina a partir de la posición de micromemoria que le corresponda.

c) Detallar el número de palabras y el contenido de la ROM de Proyec-

ción teniendo en cuenta el código de operación de cada instrucción de la tabla 4.12. En total, la Memoria de μPrograma contiene 64 posiciones, aunque en la tabla 4.14 aparezcan menos.

Instrucción	OpCode	Tarea a realizar
ADD A,B	1111	$B \Leftarrow A + B$
MOVE A,B	1101	$B \Leftarrow A$
MOVE (A+B),B	1011	$B \Leftarrow MEM[A + B]$
BGT C	1001	Si (flag_N=0 Y flag_Z=0) $\Rightarrow PC \Leftarrow C$

Tabla 4.12: Conjunto de instrucciones (Prob. 6)

Señal	Significado
c1, c2	Selección de registro para lectura al bus X y para escritura.
c3, c4	Selección de registro para su volcado al bus Y .
c5	Permiso de salida al bus X del registro seleccionado por $c1, c2$.
c6	Permiso de escritura del resultado de la ALU en el registro seleccionado por $c1, c2$.
c7	Carga del registro Acumulador (AC).
c8	Carga del registro de Instrucciones (RI).
c9, c10	Escritura y lectura, respectivamente, del registro DR.
c11, c12	Selecciona la operación al realizar en la ALU: 00=Suma. 01=Resta. 10=Producto. 11=División.
c13	$Carry$ de entrada a la ALU.
c14	Vuelca el resultado de la ALU al Bus de Datos.
c15	Actualiza los flags del registro de estado.
c16	Debe estar a 1 cada vez que se utilice la memoria (Chip Select).
c17	1 = Lectura de memoria a DR. 0 = Escritura en memoria de DR.
c18	Permite volcar el campo “Dir. de Salto” de la microinstrucción al Bus de Datos.
c19	Controla el Multiplexor-1.

Tabla 4.13: Puntos de control (Prob. 6)

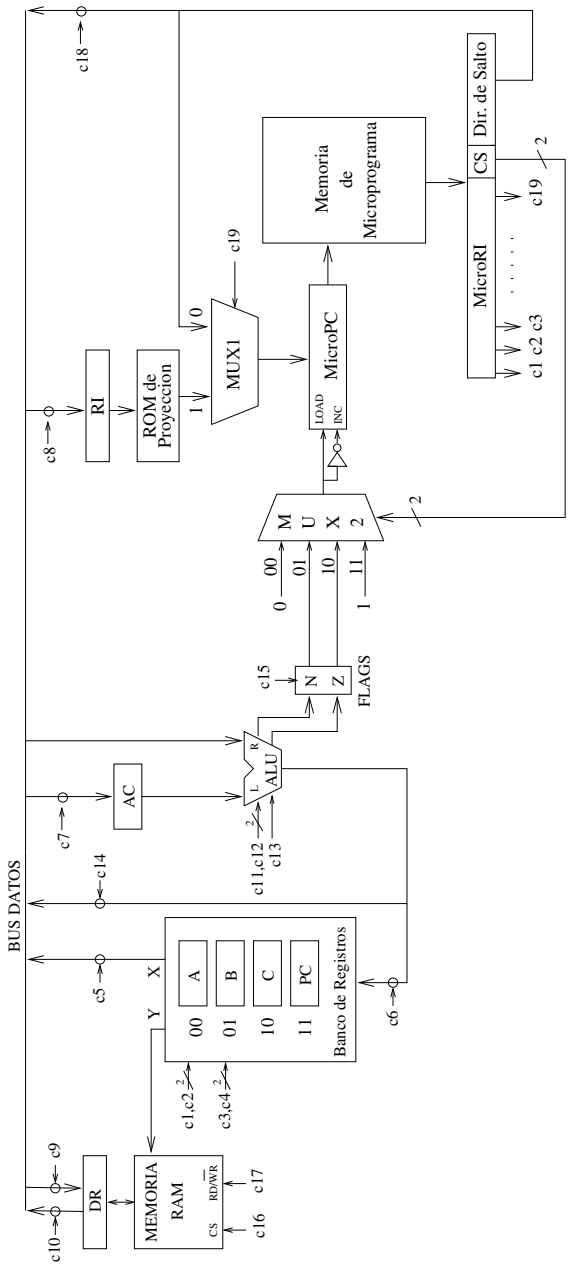


Figura 4.25: Unidades de Proceso y de Control (Prob. 6)

		Banco de Reg.					AC, RI, DR				ALU				SR	Mem.		Cte	M	Salto	
Comentario	Dir.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	CS	Dir. Salto	.
	0																						
	1																						
	2																						
	3																						
	4																						
	5																						
	6																						
	7																						
	8																						
	9																						
	10																						
	11																						
	12																						
	13																						
	14																						
	15																						
	16																						
	17																						
	18																						
	19																						
	20																						

Tabla 4.14: Memoria de microprograma (Prob. 6)

7. La figura 4.26 representa la CPU de un microprocesador con una unidad de control microprogramada. La descripción de los puntos de control de la sección de datos se incluye en la tabla 4.16. Como características más interesantes de esa sección de datos cabe destacar:
   - **Memoria RAM** externa para datos e instrucciones de 64Kx16 bits (=128Kbytes). La memoria se direcciona directamente desde alguno de los registros de dirección del banco de registros de dirección. El dato que se lee de la memoria puede ir a cualquiera de los registros de cualquier banco, `IR`, `TEMPA` o `TEMPB`. El dato que se escribe en memoria debe venir de algún registro de uno de los bancos o de la `ALU`.
   - **Banco de 4 registros de dirección**, de 16 bits cada uno. Este banco incluye el `PC`, el registro índice `SI`, el registro base `B` y un registro temporal de direcciones `TMP`. `c3-c4` son las señales que controlan la selección del registro de dirección que direcciona la memoria y que puede volcar su contenido al BUS_CPU (si `c15`=1), como se indica en la tabla 4.15 (a). El punto de control `c5` decide si se va a leer o a escribir en el registro seleccionado.

c3	**c4**	Registro de dirección
0	0	`PC`
0	1	`SI`
1	0	`B`
1	1	`TMP`

(a)

c6	**c7**	**c8**	Registro de datos
0	0	X	`Rx`
1	0	X	`Ry`
X	1	0	`R0`
X	1	1	`R1`

(b)

Tabla 4.15: (a) Selección del registro de dirección; (b) Selección del registro de datos

   - **Banco de 16 registros de datos** (`R0:R15`), de 16 bits cada uno. Están direccionados desde el campo correspondiente de la instrucción máquina, bien `Rx` o bien `Ry`, como muestra la tabla 4.17. De estos registros, el contenido de `R0` y `R1` viene fijado por el HW a $\texttt{0000}_{(16}$ y $\texttt{0001}_{(16}$ respectivamente y no son modificables por el programador. Las señales que controlan la selección del registro de datos son `c6-c8` según se indica en la tabla 4.15 (b). Como se puede ver, si `c7=0` el campo `Rx` (si `c6=0`) o el campo `Ry` (si `c6=1`) del Registro de Instrucción (`IR`) son los que seleccionan el registro del banco

de registros de datos. Sin embargo, si `c7=1`, el punto de control `c8` seleccionará el registro `R0` o `R1`, independientemente del contenido de los campos `Rx` o `Ry`.

- **Registro de instrucción** (`IR`) de 16 bits. El formato de instrucción consta de: un campo código de operación `C.O.` de 4 bits, un campo no usado, `NU`, de 4 bits, un campo registro de datos `Ry` de 4 bits y un campo registro de datos `Rx` de 4 bits. Las instrucciones pueden especificar 1 ó 2 operandos explícitamente. El hecho de que los registros de datos sean seleccionados desde el `IR` tiene la ventaja de no necesitar microrutinas distintas para instrucciones que realizan la misma operación sobre registros de datos distintos. Por ejemplo, la microrutina asociada a la instrucción `MOVE R2,R3`, es la misma que la microrutina para `MOVE R5,R2` ya que el C.O. de la instrucción será el mismo y la selección de los registros vendrá especificada en los campos `Rx` y `Ry` de `IR`.
- **ALU con sumador/restador** asociado a 2 flags de estado, carry (`C`) y cero (`Z`).

En este diseño se suponen resueltos los problemas de temporización y, dado que la carga de los registros se realiza al final del ciclo máquina, se requiere 1 ciclo de reloj para:

- un acceso de lectura de la memoria y almacenamiento en registro o lectura de registro y escritura en memoria RAM.
- una operación con la `ALU` y transferencia del resultado a algún registro.
- una transferencia entre registros de bancos diferentes.
- una transferencia entre algún registro de uno de los bancos y cualquiera de los registros temporales de la `ALU` (`TEMPA`, `TEMPB`).

Además, nos aseguran que la actualización del registro de estado tiene lugar antes que la carga del contador de microprograma. El registro de estado sólo se debe modificar en las instrucciones aritméticas o lógicas.

En el problema de diseño se pide:

a) Diseñar la microrrutina de *fetch* y decodificación. Describirla en *pseudocódigo* y detallar el contenido de la memoria de microprograma en binario a partir de la dirección `0`. Utiliza la tabla 4.18 para detallar las microinstrucciones, utilizando la columna **Descripción** para indicar en pseudocódigo el significado de cada microinstrucción.

b) Diseñar las microrrutinas asociadas a cada una de las instrucciones

de la tabla 4.17. Descríbelas en *pseudocódigo* y detalla el contenido de la memoria de microprograma en binario, colocando cada microrrutina a partir de la posición de micromemoria que le corresponda.

c) Detallar el número de palabras de la `ROM de Proyección` y el contenido de las posiciones de esta ROM que conozcas teniendo en cuenta el `C.O.` de cada instrucción de la tabla 4.17.

Señal	**Acción de control**
c1	Chip Select de la memoria RAM.
c2	1=Escritura en RAM; 0=Lectura en RAM.
c3, c4	Selección del registro de dirección según la tabla 4.15 (a).
c5	0=Lectura de reg. de dir.; 1=Escritura en reg. de dir. con el dato del `BUS_CPU`.
c6, c7, c8	Selección del registro de datos según la tabla 4.15 (b).
c9	0=Lectura de reg. de datos; 1=Escritura en reg. de datos con el dato del `BUS_CPU`.
c10	Carga `TEMPA` con el dato del `BUS_CPU`.
c11	Carga `TEMPB` con el dato del `BUS_CPU`.
c12	Carga el registro de estado.
c13	Carga `IR` con el dato del `BUS_CPU`.
c14	Operación con la `ALU`: 1=A-B; 0=A+B.
c15	Vuelca al `BUS_CPU` el contenido del registro de dirección seleccionado.
c16	Vuelca al `BUS_CPU` el contenido del registro de datos seleccionado.
c17	Vuelca al `BUS_CPU` el resultado de la `ALU`.

Tabla 4.16: Puntos de control de la sección de datos (Prob. 7)

Instrucción	**Formato de la instrucción**				**Tarea**
	`C.O.`	`NU`	`Ry`	`Rx`	
`MOV Ry,Rx`	0011	XXXX	`Ry`	`Rx`	`Ry` → `Rx`
`SUB Ry,Rx`	0000	XXXX	`Ry`	`Rx`	`Rx-Ry` → `Rx`
`BCC Ry`	0101	XXXX	`Ry`	XXXX	Si `C=0`, `Ry` → `PC`
`MOVE (SI+Ry),Rx`	0001	XXXX	`Ry`	`Rx`	`MEM[SI+Ry]` → `Rx`

Tabla 4.17: Conjunto de instrucciones (Prob. 7)

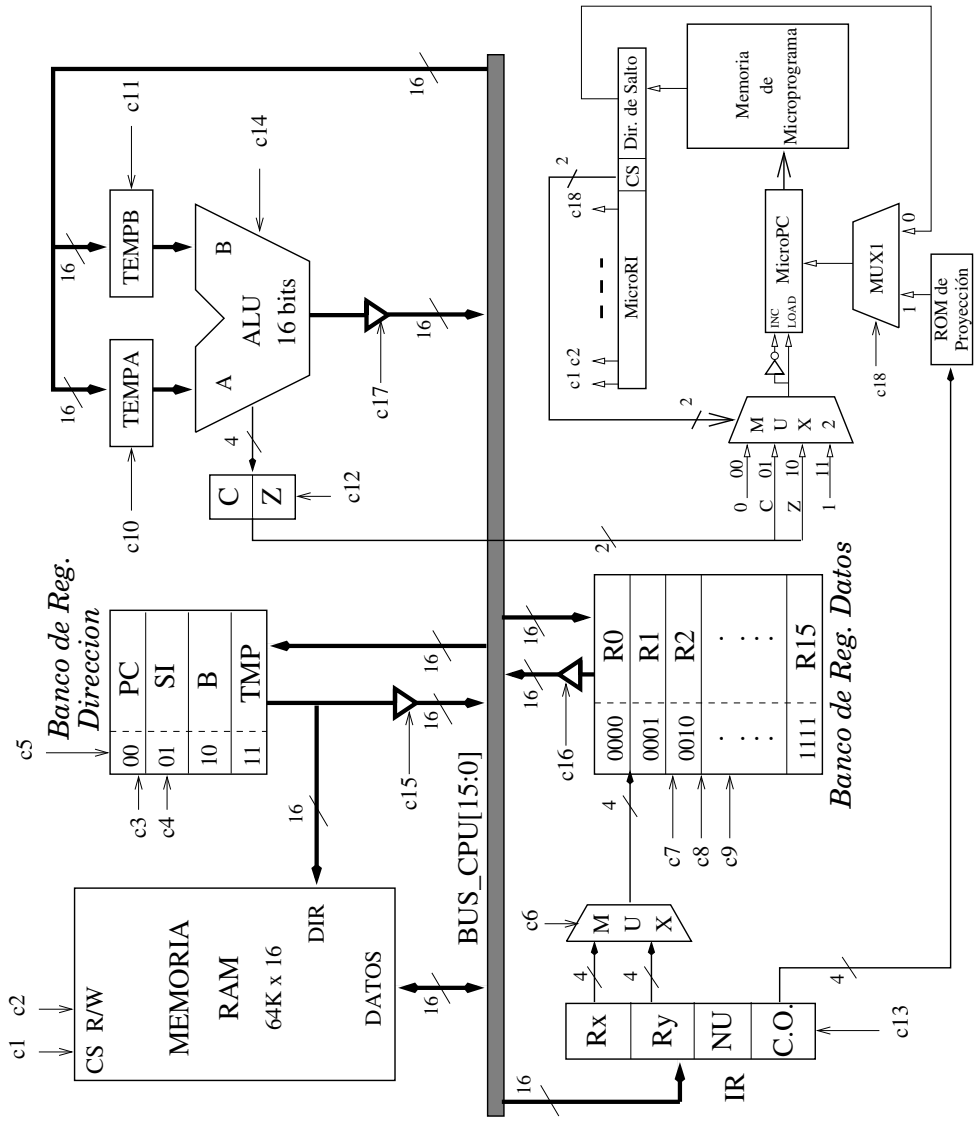


Figura 4.26: Unidades de Proceso y de Control (Prob. 7)

Descripción	Dir.	Mem.	Reg. Dir.	Reg. Dat.	Carga	ALU	Transf.	M	CS	Salto
		c1 c2	c3 c4 c5	c6 c7 c8 c9	c10 c11 c12 c13	c14	c15 c16 c17	c18		Dir. Salto
	0									
	1									
	2									
	3									
	4									
	5									
	6									
	7									
	8									
	9									
	10									
	11									
	12									
	13									
	14									
	15									
	16									

Tabla 4.18: Memoria de microprograma (Prob. 7)

8. Se pretende diseñar la Unidad de Control Microprogramada para el procesador mostrado en la figura 4.27. La Unidad de Datos contiene los registros PC e IR, un Banco de registros (B) con 256 registros de 8 bits, todos ellos accesibles a través de un registro BAR (que se utilizará para almacenar el número de registro al que se quiere acceder, bien para leer su contenido o bien para modificarlo). El registro BDR se utiliza como registro intermedio para el almacenamiento temporal de los datos relacionados con alguno de los registros del banco. Esta CPU también dispone de un registro de estados con 2 flags (Z y N); una ALU con 2 operaciones distintas y 3 registros de propósito general (L y R a su entrada, y TMP a su salida); una Memoria (M) de 256 palabras de 8 bits cada una, que contiene tanto instrucciones como datos y tiene asociado el registro MAR para las direcciones, no necesitando ningún registro intermedio para almacenar temporalmente los datos, sino que son volcados directamente sobre el BUS principal del sistema, pudiendo almacenarse dicho valor en cualquier registro directamente conectado con el BUS. El tamaño de palabra de la ALU, del Banco de Registros (B) y de la Memoria Principal (M) es de 8 bits.

   Es importante resaltar que cada instrucción está almacenada en 1 o varias palabras de memoria, con lo que posiblemente será necesario realizar más de un acceso a memoria para acceder a la instrucción completa. El juego de instrucciones de este procesador se caracteriza por la peculiaridad de que la primera palabra de la instrucción es siempre el codigo de operación (COP) de la misma, que es por tanto de 8 bits. El resto de palabras de la instrucción son opcionales (depende de cada instrucción concreta) y se corresponden a números de registro utilizados en la ejecución de la instrucción. Estos operandos son tambien de 8 bits (ver figura 4.27).

   El conjunto de instrucciones del procesador junto con sus códigos de operación y su significado, asi como el número de palabras de memoria de las que se compone (P) y el orden de acceso se muestra en la tabla 4.19. Los puntos de control que gobiernan el funcionamiento del data-path se describen en la tabla 4.20.

   Se pide:

   a) Diseñar la Unidad de Control de dicho procesador, especificando claramente el formato de la microinstrucción utilizado y los nuevos puntos de control si fueran necesarios. Se debe utilizar microprogramación horizontal.

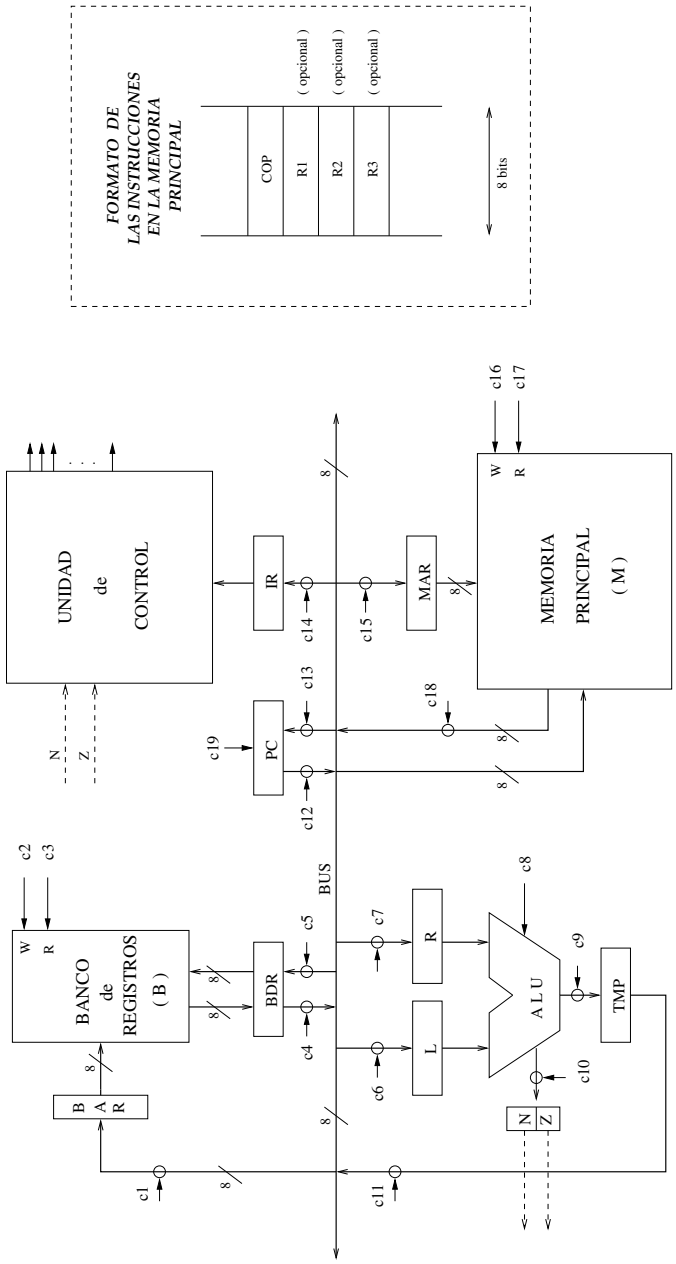


Figura 4.27: Procesador microprogramado (Prob. 8)

Nemotécnico	COP	Significado	P	Orden
SALTA Z R1	00001111	si Z=1, PC ← B(R1)	2	(COP, R1)
MUEVE R1,R2	00001010	B(R2) ← B(R1)	3	(COP, R1, R2)
SUMA R1,R2,R3	00000101	B(R3) ← B(R1)+B(R2)	4	(COP, R1, R2, R3)

Tabla 4.19: Conjunto de instrucciones del procesador (Prob. 8)

Señal	Nemotécnico	Significado
c1	loadBAR	Carga en BAR el contenido del BUS
c2	writeB	Escritura en el banco de registros (B(BAR) <- BDR)
c3	readB	Lectura del banco de registros (BDR <- B(BAR))
c4	vuelcaBDR	Vuelca el contenido de BDR en el BUS
c5	loadBDR	Carga en BDR el contenido del BUS
c6	loadL	Carga en L el contenido del BUS
c7	loadR	Carga en R el contenido del BUS
c8	ALU	0: L+R, 1: L+1
c9	loadTMP	Carga del registro TMP con el resultado de la ALU
c10	flags	Carga el registro de estado del procesador
c11	vuelcaTMP	Vuelca el contenido de TMP al BUS
c12	vuelcaPC	Vuelca el contenido de PC al BUS
c13	loadPC	Carga el PC con el contenido del BUS
c14	loadIR	Carga el IR con el contenido del BUS
c15	loadMAR	Carga el MAR con el contenido del BUS
c16	writeM	Escritura en memoria
c17	readM	Lectura de memoria
c18	vuelcaM	Vuelca el contenido de la memoria al BUS
c19	incPC	Incremento del Contador de Programa

Tabla 4.20: Puntos de control del procesador (Prob. 8)

b) Escribir el microcódigo correspondiente a la rutina común de búsqueda y decodificación, así como las microrrutinas asociadas a cada una de las instrucciones máquina mostradas en la tabla 4.19.

c) Si se ha utilizado en el diseño de la Unidad de Control, describir internamente el Sistema de Proyección.

NOTA: Suponer resueltos todos los posibles problemas de temporización, y que tanto las operaciones de transferencia entre 2 registros cualquiera del procesador, cualquier operación con la ALU o acceso a memoria (lectura/escritura) o al banco de registros (lectura/escritura) requieren un único ciclo de reloj.

9. Sean las secciones de datos y de control microprogramado de la figura 4.28. Sus elementos más destacados son:
   - **Memoria RAM** externa de 2^{32} celdas de 16 bits (4 Giga-palabras), común para datos e instrucciones. Está asociada a un registro de datos, `DR` de 16 bits, y otro de direcciones `MAR` de 32 bits.
   - **Un banco de ocho registros** entre los que se encuentran seis registros de propósito general, `D0-D5` y un registro índice `IX` dividido en 2 partes: `IXL`, que almacena los 16 bits menos significativos, e `IXH`, que almacena los 16 bits más significativos de una dirección de 32 bits. Las señales de control `c1`, `c2` y `c3`, seleccionan uno de los ocho registros para volcar su contenido al `BUS` (si `c5` = 1), o para cargarse con el dato del `BUS` (si `c4` = 1).
   - **Registros temporales** (`TMP1` y `TMP2`), que pueden ser cargados desde el `BUS`, y proporcionan los operandos de la `ALU`.
   - **Registro de Instrucción** (`IR`) de 16 bits que almacena una instrucción con un código de operación (`OpCode`) de 16 bits. Todas las instrucciones del micro ocupan una palabra en memoria.
   - **Unidad Aritmético-Lógica** de 16 bits capaz de realizar 4 operaciones controladas por las señales `c9` y `c10`. Notar que la `ALU` proporciona el carry `C` y que éste a su vez realimenta la entrada de carry `Cin` de la `ALU`.
   - Notar también que *sólo* `PC` *y* `MAR` *son registros de 32 bits.* El resto de los registros son de 16 bits.

   Suponer resueltos los posibles problemas de temporización y, dado que los registros se cargan al final del ciclo máquina (incluído el contador de microprograma), es necesario:
   - 1 ciclo de reloj para completar la siguiente cadena de operaciones: "Volcado de alguno de los registros `TMP1` o `TMP2` a la `ALU`"; "Operación `ALU`", y "Escritura en alguno de los registros".
   - 1 ciclo para cualquier transferencia entre dos registros cualesquiera (`PC`, `MAR`, `D0`, `IXL`, `TMP1`, ...)
   - 1 ciclo de reloj para la lectura o escritura del dato almacenado en `DR` en la posición de memoria dada por `MAR`.

   Se pide:

   a) Diseñar la Unidad de Control microprogramada. Justificar el tamaño del registro de microinstrucción.

   b) Diseñar la microrrutina común de búsqueda y decodificación. Utilizar la tabla 4.23 para codificar las microinstrucciones. En la colum-

na **Descripción** detallar en RTL el significado de cada microinstrucción. *Notar que esta tabla no está completa.*

c) Diseñar las microrrutinas asociadas a cada una de las instrucciones de la tabla 4.22.

d) Detallar el número de palabras de la `ROM de Proyección` y el contenido de las posiciones que conozcas.

SEÑAL	**Significado**
c1, c2, c3	Selección de registro para lectura/escritura al `BUS`.
c4	Carga del dato del `BUS` al registro seleccionado por `c1,c2, c3`.
c5	Volcado del registro seleccionado por `c1,c2,c3` al `BUS`.
c6	Carga del registro `TMP1` con el dato del `BUS`.
c7	Carga del registro `TMP2` con el dato del `BUS`.
c8	Reset del registro `TMP2`.
c9, c10	Selecciona la operación a realizar en la `ALU`: 00=`L+R`, 01=`L-R`, 10=`L+R+1`, 11=`L+R+C`.
c11	Carga del registro de estado.
c12	Vuelca la salida de la `ALU` al `BUS`.
c13	Carga los 16 bits menos significativos de `MAR` con el dato del `BUS`.
c14	Carga los 16 bits mas significativos de `MAR` con el dato del `BUS`.
c15	Carga del registro `DR` con el dato del `BUS`.
c16	Vuelca el registro `DR` en el `BUS`.
c17	1 = Habilita acceso a RAM; 0 = Deshabilita acceso a RAM.
c18	1 = Lectura de RAM; 0 = Escritura en RAM.
c19	Carga los 16 bits menos significativos de `PC` con el dato del `BUS`.
c20	Carga los 16 bits más significativos de `PC` con el dato del `BUS`.
c21	Vuelca los 16 bits menos significativos de `PC` al `BUS`.
c22	Vuelca los 16 bits más significativos de `PC` al `BUS`.
c23	Incrementar PC
c24	Carga del registro `IR` con el dato del `BUS`.

Tabla 4.21: Descripción de los puntos de control de la CPU (Prob. 9)

Instrucción	**OpCode**	**Tarea a realizar**
ADD D3,D4	$001F_{(16}$	`D4` ← `D3 + D4`
BCC IX	$0020_{(16}$	**IF** `(flag_C=1)` **THEN** `PC` ← `IX`
MOV D0,D1	$0000_{(16}$	`D1` ← `D0`
LDIX D5,D2	$0003_{(16}$	D2 ← `MEM[IX+D5]`

Tabla 4.22: Subconjunto de instrucciones de la CPU (Prob. 9)

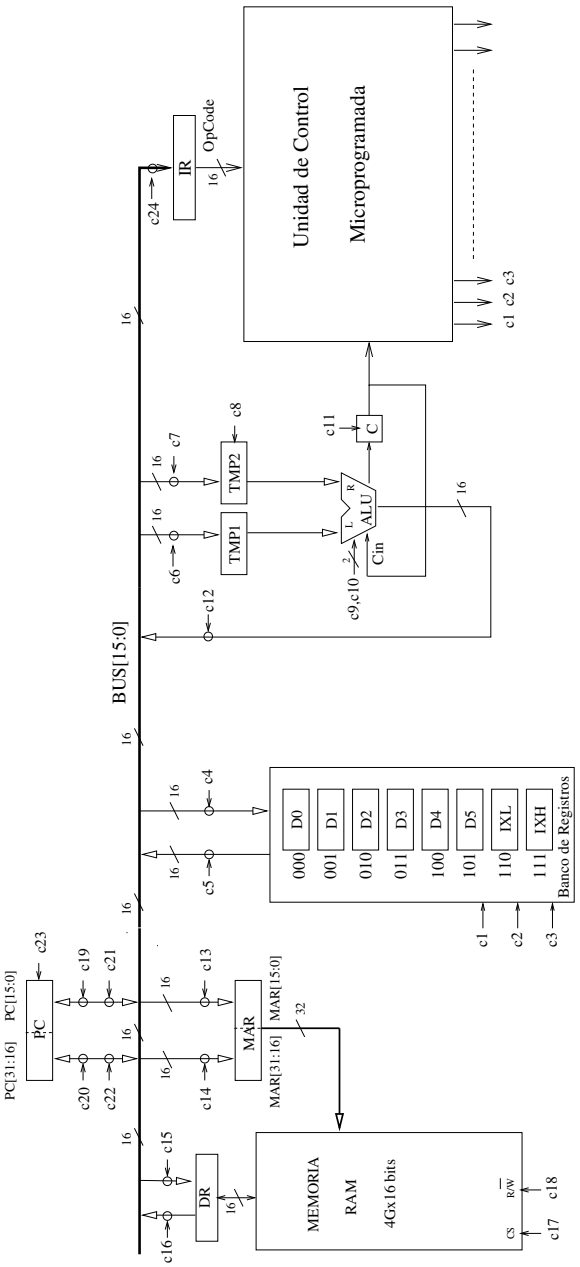


Figura 4.28: Unidades de Datos y de Control (Prob. 9)

Descripción	Dir.	Banco Regs.	TMP	ALU	MAR	DR	MEM	PC	RI	
		c1 c2 c3 c4 c5	c6 c7 c8	c9 c10 c11 c12	c13 c14	c15 c16	c17 c18	c19 c20 c21 c22 c23	c24	

Tabla 4.23: Memoria de microprograma (Prob. 9)

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

10. Diseñar la unidad de control cableada siguiendo los dos métodos explicados para el procesador planteado en la sección 4.4 extendiendo el conjunto de instrucciones con las que aparecen en la tabla 4.24.

Mnemónico	Descripción
COMP	Complemento del acumulador $not(ACC) \rightarrow ACC$
JMP A	Salto incondicional a la posición A: $A \rightarrow PC$
SHIFTR	Desplazamiento de un bit a la derecha del acumulador: $RightShift(ACC, 1) \rightarrow ACC$
AND A	Y lógico del acumulador y el contenido de la posición A: $ACC \wedge (A) \rightarrow ACC$
STOREI #N	Guarda un valor inmediato en el acumulador: $N \rightarrow ACC$

Tabla 4.24: Extensión del conjunto de instrucciones

11. Para la unidad de control de la figura 4.29 diseñada por el método de los elementos de retardo,
    a) Escribir el diagrama de flujo que implementa y el cronograma de las señales de control que genera.
    b) Representar los elementos de retardo con biestables D con CLEAR y PRESET indicando la conexión de la señal de reloj y la señal de inicio (BEGIN).
    c) Rediseñar la unidad de control utilizando el método del contador de secuencias.

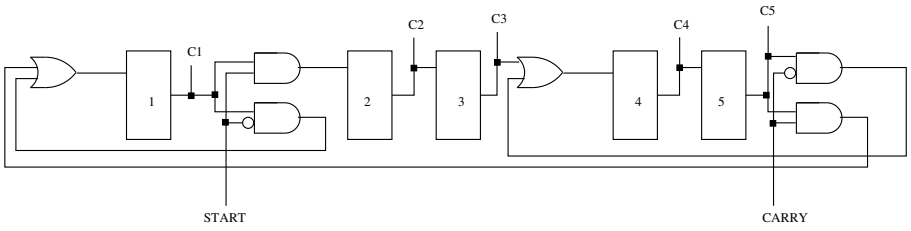


Figura 4.29: Unidad de Control (Prob. 11)

12. La figura 4.31 representa un procesador con control cableado cuyos puntos de control se detallan en la tabla 4.25. Podemos destacar del procesador:

- Memoria RAM de 32K palabras de 16 bits. La memoria se estructura en palabras de dos bytes, y su direccionamiento se realiza *a nivel de byte*.
- El tamaño de todos los registros existentes es de 1 palabra, así como el del bus de datos que se ha denotado con $busD$. Dado el diagrama propuesto se observa que para almacenar un dato en $OP2$ ha de ser cargado previamente en $OP1$. El registro acumulador ACC se puede usar como registro temporal si fuera necesario.
- La ALU permite 4 operaciones: $OP1 + OP2$, $OP1 - OP2$, $OP2 + 0$, y $OP2 + 1$, generando un bit de estado de cero que puede ser almacenado en FZ.
- Se considera que un solo ciclo de reloj es suficiente para leer o escribir de memoria así como para transferir datos entre registros conectados. No es necesario considerar en el ejercicio la temporización de los puntos de control activos por flanco.
- Todas las instrucciones ocupan 2 palabras de memoria. La primera palabra es el código de operación y la segunda contiene potencialmente un operando. Aquellas instrucciones sin operando desaprovechan, pues, esta segunda palabra ya que no la usan. Las instrucciones ocupan siempre, un número par de bytes, y por ello `PC` puede ser incrementado de 2 en 2 a través de un sumador auxiliar.

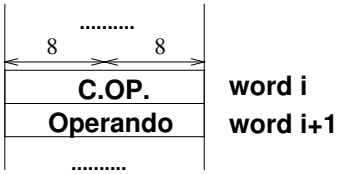


Figura 4.30: Formato de las instrucciones

Sobre el procesador descrito se pide:

a) Genera el diagrama de estados (flujo) de la unidad de control **cableada**, capaz de ejecutar el conjunto de instrucciones reducido que se muestra en la siguiente tabla. En cada estado ha de indicarse la operación de transferencia de registro que corresponda, así como los puntos de control que se activan. Debe considerarse que la búsqueda y decodificación es común a las instrucciones. Observa que un decodificador específico *Inst-DEC* activa una señal correspondiente a la instrucción fue cargada en IR.

Mnemónico	Operando	Operación	Significado
`DUP A`	`A`	`ACC ← 2*A`	Duplica A y lo guarda en el acumulador
`JIZ bdir`	`bdir`	`Si FZ=1 ⇒ PC ← bdir`	Salto al byte bdir, condicionado a FZ.
`JEQ [bdir]`	`bdir`	`Si OP1=OP2 ⇒ PC ← Mem(bdir)`	Salto condicional indirecto a la palabra contenida en bdir cuando OP1=OP2.
`JMP Wdir`	`Wdir`	`PC ← 2*Wdir`	Salto incondicional a la palabra Wdir.

b) Construir la unidad de control cableada, según la metodología de los **elementos de retardo** y escribir las ecuaciones de los puntos de control correspondientes.

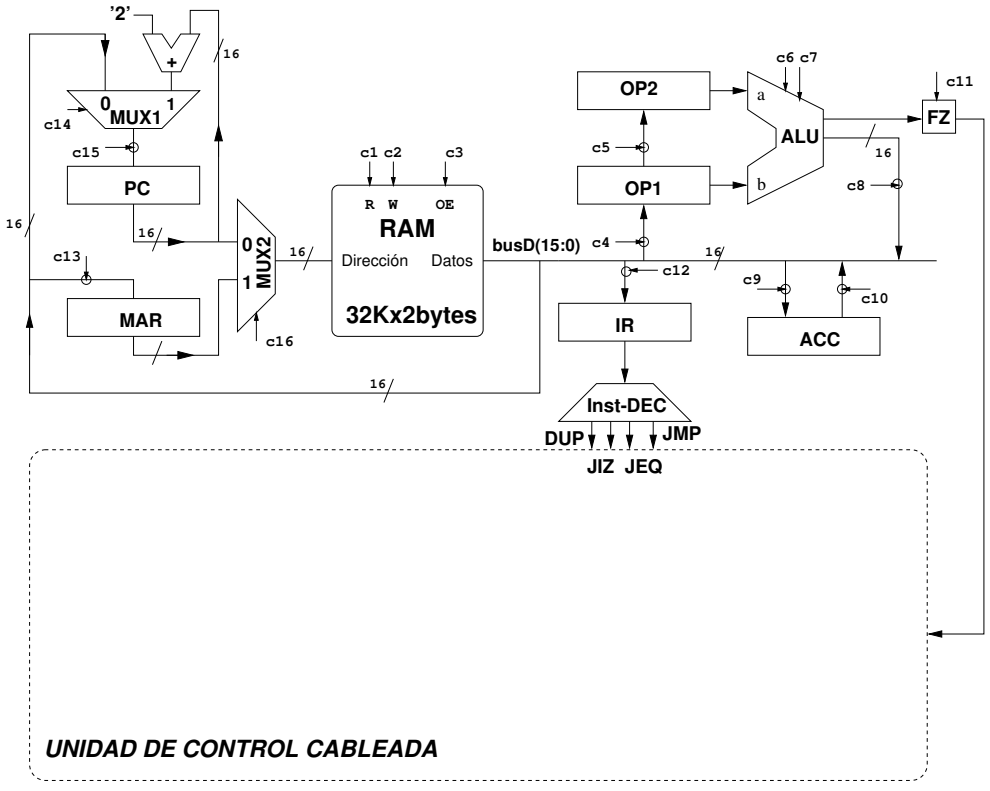


Figura 4.31: Datapath (Prob. 12)

P.C.	Función	Ecuación
c1	Habilitación de lectura de RAM	c1=
c2	Habilitación de escritura en RAM	c2=
c3	Habilita salida de RAM	c3=
c4	Carga de OP1: OP1 ← busD	c4=
c5	Carga de OP2: OP2 ← OP1	c5=
c6,c7	Operación ALU: 00:a+b, 01:a-b, 10:a+0, 11:a+1	c6= c7=
c8	Abre conexión entre ALU y busD	c8=
c9	Carga: ACC ← busD	c9=
c10	Volcado de ACC en busD	c10=
c11	Carga del bit de estado FZ	c11=
c12	Carga: IR ← busD	c12=
c13	Carga: MAR ← busD	c13=
c14	Selección MUX1 (0:busD, 1:PC+2)	c14=
c15	Carga PC con la salida de MUX1	c15=
c16	Selección MUX2 (0:PC, 1:MAR)	c16=

Tabla 4.25: Puntos de control (Prob. 12)

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

5 Sección de Procesamiento: Algoritmos Aritméticos

OBJETIVOS

- Describir brevemente el HW y los algoritmos encargados de implementar las operaciones aritméticas en la ALU
- Introducir la aritmética en punto flotante a partir de los algoritmos de aritmética entera

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

5.1. ARITMÉTICA DE PUNTO FIJO

Vamos a comenzar este tema estudiando la aritmética en punto fijo, de la que los enteros son el caso particular de estar situado el punto decimal a la derecha del dígito menos significativo del número. Veremos tanto el HW que hay dentro de la ALU como los algoritmos que se tienen que ejecutar para realizar las operaciones aritméticas: **suma**, **resta**, **multiplicación** y **división**. Además cuando se estudie la aritmética en punto flotante se podrá comprobar como ésta se basa en operaciones en punto fijo.

5.2. SUMA Y RESTA

Orientaremos esta sección desde dos puntos de vista. El primero será el punto de vista *software*, en el que veremos qué algoritmos son necesarios aplicar para realizar estas operaciones, en función del convenio de representación de los operandos (binario natural, Signo/Magnitud, Complemento a 1 o Complemento a 2). Será después cuando veamos las posibles implementaciones *hardware* y una comparación entre ellas, atendiendo a las características, casi siempre en compromiso, de coste y velocidad.

5.2.1. Algoritmos de suma y resta

Binario natural

Si nos detenemos a pensar en la forma en que se realiza con lápiz y papel la suma de dos números A y B de **n** bits, observamos que para la obtención del dígito i-ésimo del resultado, s_i, intervienen no sólo los dígitos a_i y b_i de los sumandos, sino, además, el carry c_i, que ha sido generado previamente (en la etapa **i-1**). Además, junto con s_i, se va a generar el carry c_{i+1}, necesario para el cómputo de s_{i+1}:

$$\begin{array}{cccccccc} & c_{n+1} & c_n & \dots & c_3 & c_2 & c_1 & = C \\ & & a_n & \dots & a_3 & a_2 & a_1 & = A \\ + & & b_n & \dots & b_3 & b_2 & b_1 & = B \\ \hline & s_{n+1} & s_n & \dots & s_3 & s_2 & s_1 & = S \end{array} \tag{5.1}$$

Es decir, el *carry* c_i se genera en la etapa **i-1** y se suma en la etapa **i**. Suponemos el carry inicial de entrada, c_1, inicializado desde una de las entradas de control del sumador, de forma que se pueda poner a 0 ó a 1. Como podemos

ver de la suma anterior, $s_{n+1} = c_{n+1}$. Si la aritmética es de **n** bits, es decir, los números han de ser representados con **n** bits, se produce *overflow* si $s_{i+1} = 1$.

Las ecuaciones lógicas necesarias para obtener s_i y c_{i+1} son:

$$\begin{aligned} s_i &= (a_i \oplus b_i) \oplus c_i \\ c_{i+1} &= a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i \end{aligned} \tag{5.2}$$

La realización de un sumador de 1 bit seguirá la siguiente tabla de verdad (tabla 5.1), en la que se considera, junto con los dos dígitos binarios a sumar, el acarreo procedente de la etapa anterior, generándose, además del dígito de la suma, el acarreo para la etapa siguiente:

Entradas			**Salidas**	
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Tabla 5.1: Tabla de verdad de la suma

En este apartado hemos tratado la suma de números positivos representados en binario natural con **n** bits. Si además queremos realizar operaciones con números negativos, debemos utilizar los sistemas de representación en Signo/Magnitud, Complemento a 1 o Complemento a 2.

Signo/Magnitud

Bajo este convenio de representación de enteros negativos, los bits a_n y b_n de la expresión 5.1 representan el bit de signo de los sumandos y s_n el bit de signo de la suma. El algoritmo para realizar la **suma** es:

- Si $a_n = b_n \rightarrow$ Sumandos del mismo signo.
  1. Se suman las magnitudes de **n-1** dígitos (bits) mediante la ecuación 5.2.
  2. Poner $s_n = a_n$.
  3. Si $c_n = 1 \Rightarrow$ Overflow.

- Si $a_n \neq b_n \rightarrow$ Sumandos de distinto signo.
  1. Se restan las magnitudes de **n-1** (bits) mediante la ecuación:

$$\begin{aligned} s_i &= (a_i \oplus b_i) \oplus c_i \\ c_{i+1} &= \overline{a_i} \cdot b_i + (\overline{a_i} \oplus b_i) \cdot c_i \end{aligned} \tag{5.3}$$

     donde c_i es el acarreo de la resta, también llamado *borrow*.
  2. Si

$$c_n = \begin{cases} 0 \Rightarrow & \text{Poner } s_n = a_n, \\ 1 \Rightarrow & \text{Poner } s_n = b_n \text{ y hacer C2}(s_{n-1} \ldots s_2\, s_1). \end{cases}$$

Para **restar** $A - B$, habrá que cambiar el bit de signo de B ($b_n = \overline{b_n}$) y aplicar el algoritmo de suma comentado anteriormente.

Los inconvenientes de representar los números en S/M son claros:

- El algoritmo a seguir es complejo.
- El coste asociado al HW necesario es elevado, dado que necesitamos una unidad sumadora que implemente al ecuación 5.2 y una unidad restadora que haga lo propio con la ecuación 5.3. Además, es necesario un HW para hacer el C2 para el caso en que $c_n = 1$.

Complemento a 1

Si los sumandos están representados en un sistema de numeración en C1 de **n** bits, el algoritmo de **suma** es el siguiente:

1. Sumar los operandos mediante la ecuación 5.2, sin distinguir el bit de signo del resto de los **n** bits.
2. Si $c_{n+1} = 1$ se suma 1 al resultado. Es decir, después de operar, se añade el acarreo de salida.
3. Si los valores sumados son de igual signo[9] hay *overflow* si el signo del resultado es distinto del de los operandos. Otra forma de determinar si hay *overflow* es mediante la ecuación $c_{n+1} \oplus c_n$, que tomará el valor uno en caso de que éste se produzca.

En este caso, el algoritmo para **restar** $A - B$ consiste en sumar $A + C1(B)$.

Este algoritmo tiene la ventaja, frente al de suma/resta en S/M, de no necesitar un HW específico de resta (no hay que implementar la ecuación 5.3). Pero aún podría ser más sencillo si nos pudiésemos evitar la suma del acarreo, necesaria en el algoritmo comentado.

[9] Si los sumandos son de distinto signo no puede haber *overflow*.

Complemento a 2

Si los sumandos están representados en un sistema de numeración en C2 de **n** bits, el algoritmo de **suma** es el siguiente:

❶ Sumar los valores mediante la ecuación 5.2, sin distinguir el bit de signo del resto de los **n** bits .
❷ Se desprecia el acarreo de salida (c_{n+1}).
❸ Si los operandos son de igual signo, hay *overflow* si el signo del resultado es distinto del de los sumandos. Al igual que en el algoritmo para C1, hay *overflow* si $c_{n+1} \oplus c_n = 1$.

De forma análoga a cuando trabajamos con la representación en $C1$, para **restar** $A - B$ bastará con sumar $A + C2(B)$

Este algoritmo resulta el más rápido de todos, ya que es sencillo y el acarreo se desprecia.

Por supuesto, también existen algoritmos de suma/resta para otros sistemas de representación de los números, como BCD natural, BCD exceso a 3, etc, pero se apartan del ámbito de este curso.

5.2.2. Implementación de un sumador

Como se ha visto, cuando usamos los sistemas de numeración binaria en complemento a 1 y complemento a 2, se hace innecesaria la distinción entre adición y sustracción. La resta se va a poder implementar realizando unas modificaciones mínimas sobre el elemento sumador, por lo que en este apartado nos centraremos en el estudio de los sumadores.

Sumador serie

El **sumador elemental** (1 bit) que implementa la ecuación 5.2 se puede realizar en dos etapas. La primera consiste en la construcción de un módulo llamado *half-adder* o semi-sumador, como el de la figura 5.1, que implementa la ecuación:

$$S = A \oplus B$$
$$C = A \cdot B$$

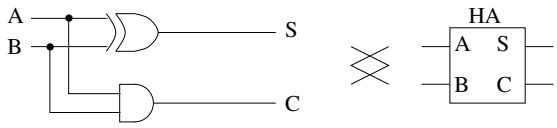


Figura 5.1: Semi-sumador

En la siguiente etapa, un **sumador completo** o *full-adder* se construye

mediante dos semi-sumadores, como se ve en la figura 5.2, que implementa la ecuación 5.2 completa.

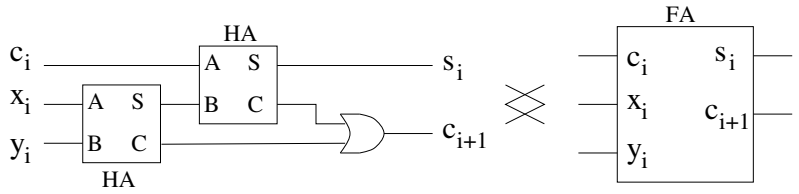


Figura 5.2: Sumador completo (de un bit)

Aunque el sumador elemental se emplea normalmente como bloque constructivo de los sumadores paralelos, que estudiaremos en el siguiente apartado, es posible construir un sumador utilizando uno sólo de esos elementos sumadores. Se tratará de un **sumador serie**, como el de la figura 5.3, en el que la suma se irá realizando bit a bit, de un modo secuencial. Para ello, A y B se pueden almacenar en sendos registros de desplazamiento, debiéndose disponer además de un biestable en el que almacenaremos el carry generado por cada etapa (suma elemental) de cara a su utilización en la siguiente.

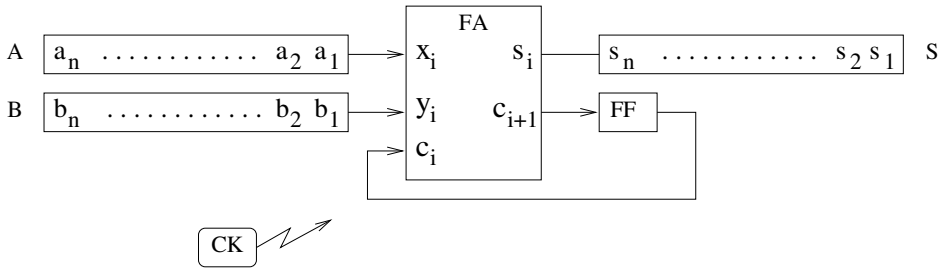


Figura 5.3: Sumador serie (de **n** bits)

Sumadores paralelos

1. **Ripple adders (Sumador con acarreo propagado o enlazado)**

   Supongamos que queremos sumar dos palabras de **n** bits. En lugar de pasar todos los bits secuencialmente a través de un simple sumador elemental, podemos presentar los **n** bits de los dos operandos en paralelo a través de **n** sumadores elementales. Estos **n** sumadores elementales están conectados de tal forma que el acarreo de salida del sumador **i**,

c_{i+1}, es el acarreo de entrada para el sumador **i+1**, con $1 \leq i \leq n-1$. Hay una línea de entrada, c_1, y una de salida, c_{n+1}. Un esquema de un sumador de este tipo para números enteros de **n** bits se muestra en la figura 5.4.

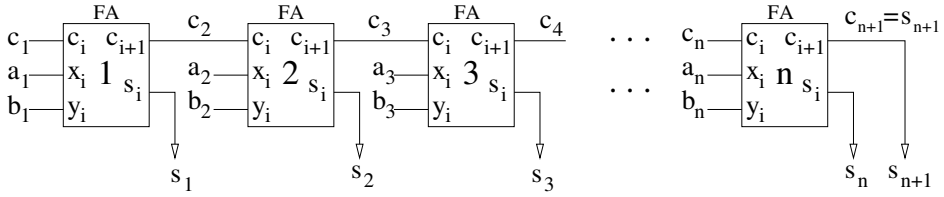


Figura 5.4: Sumador de acarreo propagado (de **n** bits)

El acarreo puede propagarse desde c_1 hasta la última suma en serie, a través de los **n** sumadores, de forma que, en el peor de los casos, el tiempo en realizar una suma corresponde a la suma de los tiempos empleados por **n** sumadores elementales. Si se diseña una ALU síncrona, este tiempo es el que debemos elegir para realizar una suma completa.

2. **Carry look-ahead adders (Sumadores con acarreo anticipado)**

Si examinamos la ecuación lógica del sumador elemental, podemos ver que la ecuación del acarreo,

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$$

puede reescribirse en función de dos términos:

a) Generación del acarreo $g_{i+1} = a_i \cdot b_i$

b) Propagación del acarreo $p_{i+1} = a_i \oplus b_i$

De forma que quedaría:

$$c_{i+1} = g_{i+1} + p_{i+1} \cdot c_i$$

Comenzando con un acarreo inicial c_1 y $g_1 = c1$, tenemos la ecuación de recurrencia:

$$c_{i+1} = g_{i+1} + \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i+1} p_k \right) \cdot g_j$$

donde Σ y Π representan las funciones lógicas OR y AND respectivamente.

Cada uno de los c_i se expresa ahora en función de a_j, b_j y c_1, con $1 \leq j \leq i$. Esto nos permite diseñar los sumadores con acarreo anticipado

(**CLA**), en donde un acarreo se genera antes de que su bit suma haya sido obtenido. Si, por ejemplo, comparamos dos sumadores comerciales de 4 bits, como el 74LS83 (con arquitectura **ripple carry**) y el 74LS283 (con arquitectura **CLA**), encontramos lo siguiente:

- Tienen aproximadamente el mismo número de puertas.
- El 74LS283 (CLA) tiene, en el peor de los casos, un retardo de 4 niveles de puertas, en lugar de los 8 que le corresponden al 74LS83.
- La velocidad del CLA es 3 veces superior a la del 74LS83

Sin embargo, restricciones eléctricas de *fan-in* y *fan-out*[10] impiden una extensión directa del sistema descrito a sumadores para palabras de gran longitud. La solución es conectar en cascada CLA's, con el acarreo de salida de uno conectado al acarreo de entrada del siguiente.

3. **Carry save adders (Sumadores con acarreo almacenado)**

Supongamos que se necesita sumar m operandos ($m > 2$). Esto ocurre, por ejemplo, en una multiplicación. La velocidad de esta operación se puede aumentar dejando la propagación del acarreo para la última suma.

Comenzaremos por describir como funciona el bloque elemental, preparado para trabajar con tres valores. Estos módulos básicos se podrán interconectar entre sí para ampliar el número de operandos, siguiendo una estructura que veremos un poco más adelante.

Cuando se va a calcular la suma de tres valores, primero sumamos los tres números sin propagar el acarreo, que sencillamente "apuntamos" en una variable, **c**, mientras guardamos los bits de suma en otra variable **s**. A continuación, realizamos $s + 2c$ para conseguir el resultado final. Por ejemplo, veamos el modo de proceder para sumar x=10, y=5 y z=12, que podríamos dividirlo en dos etapas: cálculo de **s** y **c** y obtención del resultado, $s + 2c$.

Las ecuaciones necesarias para conseguir **s** y **c** son:

$$s_i \leftarrow x_i \oplus y_i \oplus z_i$$
$$c_i \leftarrow x_i \cdot y_i + (x_i \oplus y_i) \cdot z_i$$

que son prácticamente iguales a las que implementa un *full-adder* (ecua-

[10]Por definición, el máximo número de puertas lógicas que una puerta puede excitar, permaneciendo los niveles eléctricos en los márgenes garantizados, es el **fan-out** de dicha puerta. Por analogía, el **fan-in** de una puerta es una medida de cuánto carga una de las entradas de dicha puerta a la puerta excitadora. En el circuito que implementa un CLA, la salida de algunas puertas han de conectarse a las entradas de un gran número de otras puertas, tanto mayor cuanto mayor sea el número de bits del sumador, pudiéndose alcanzar lo límites impuestos por el *fan-in* y *fan-out* de la tecnología.

ción 5.2), donde $1 \leq i \leq n$, siendo **n** el número de dígitos de los sumandos.

```
x = 10          001010
y =  5          000101        s  000011
z = 12        + 001100    + 2c  011000
                ------          ------
            s = 000011          011011 => 27
            c = 001100
```

Para implementar un *carry save adder* (**CSA**) vamos a usar **n** sumadores completos, pero en vez de conectarlos como en el caso del *ripple-adder*, tomamos como entradas los tres sumandos -x, y, z- generándose las dos salidas, s y c, como se muestra en la figura 5.5 (para el caso n=4).

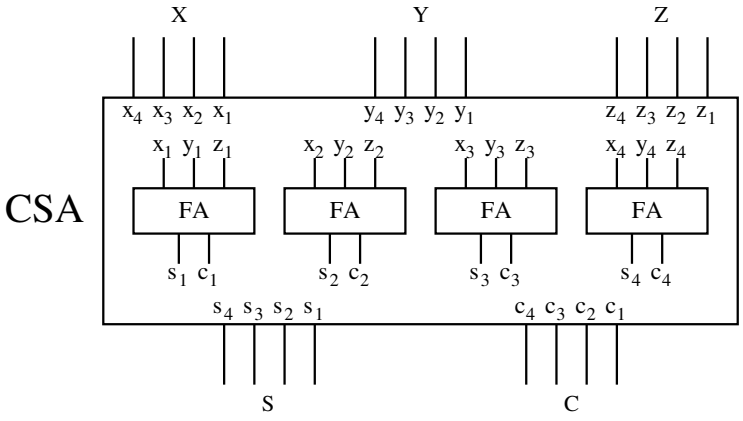


Figura 5.5: Sumador de acarreo almacenado de 4 bits

Es evidente que el tiempo de propagación de un CSA es igual al tiempo de propagación de los *full-adders* que lleva dentro, ya que los **n** sumadores completos operan en paralelo para obtener s y c.

Utilizando varios **CSA**'s interconectados, podemos realizar implementaciones del tipo *árbol de Wallace*, que nos permiten sumar un número grande de operandos, como se puede ver en la figura 5.6.

Es importante notar como el último sumador del árbol de Wallace es un sumador de acarreo propagado o anticipado de dos sumandos, que a su salida proporciona la suma total.

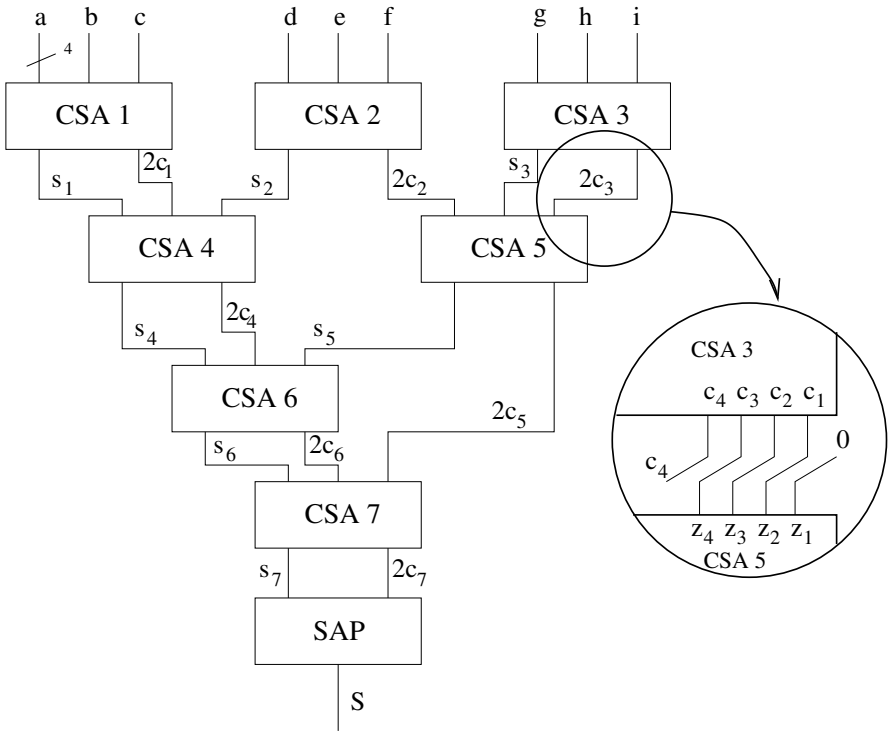


Figura 5.6: Árbol de Wallace para nueve sumandos (a, b, ..., h, i)

5.3. PRODUCTO DE ENTEROS SIN SIGNO O NATURALES

5.3.1. Introducción

Antes de formalizar los algoritmos aritméticos que vamos a implementar, recordaremos como efectuamos la multiplicación con lápiz y papel. Indicaremos qué recursos *hardware* serían necesarios para su realización como circuito y expondremos algunas transformaciones simples que reducen la complejidad de estos procesos.

Consideremos que deseamos obtener el producto $Z = Y \times X$, donde el multiplicando **Y** y el multiplicador **X** admiten la siguiente representación *Base* B con **n** dígitos.

$$X \rightarrow x_{n-1}\, x_{n-2} \ldots x_1\, x_0$$
$$Y \rightarrow y_{n-1}\, y_{n-2} \ldots y_1\, y_0$$

El procedimiento que usamos para multiplicar $Y \times X$ en un papel, se

basa en expresar el valor del multiplicador en función de los dígitos de su representación *Base B* (X es igual a la suma de los dígitos de su representación por las sucesivas potencias de la base), es decir, si tomamos $B = 10$:

$$X = x_{n-1} \cdot 10^{n-1} + \cdots + x_1 \cdot 10^1 + \cdots x_0 \cdot 10^0 \tag{5.4}$$

Al multiplicar $Y \times X$ aplicamos la propiedad distributiva en la evaluación del producto del multiplicando por la nueva expresión del multiplicador (ecuación 5.4). El cómputo se descompone en la suma de **n** factores, escalados por las sucesivas potencias de la base. Cada factor es el resultado de multiplicar un dígito del multiplicador por los **n** dígitos del multiplicando.

En la figura 5.7 mostramos un ejemplo de multiplicación de enteros en base diez siguiendo el algoritmo que utilizamos típicamente para multiplicar en un papel.

Ejemplo: $345 \times 123 = 345 \times (1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0) =$

$$(345 \times 1 \cdot 10^2) + (345 \times 2 \cdot 10^1) + (345 \times 3 \cdot 10^0)$$

```
          345  -> Y          345
        x 123  -> X        x 123
        -----              -----
      (345x3)               1035  -> PP0
     (345x2)                690   -> PP1
+ (345x1)                + 345    -> PP2
-----------              -------
                           42435
```

Figura 5.7: Ejemplo de multiplicación con lápiz y papel

Una implementación hardware del algoritmo del producto, tal y como se realiza con lápiz y papel, requeriría almacenar temporalmente los productos parciales PP_i. Sin embargo, si, inmediatamente a su obtención, sumamos el producto PP_i, acumulándolo en una única variable P, obtendremos un sustancial ahorro de recursos de almacenamiento. Para cada dígito del multiplicador obtendremos su producto parcial por el multiplicando y lo acumularemos en P. La figura 5.8 muestra la realización del mismo producto de la figura 5.7 mediante la acumulación de los productos parciales, junto con el hipotético HW de multiplicación siguiendo este nuevo algoritmo.

Algorítmicamente podemos expresar este procedimiento mediante el siguiente pseudocódigo:

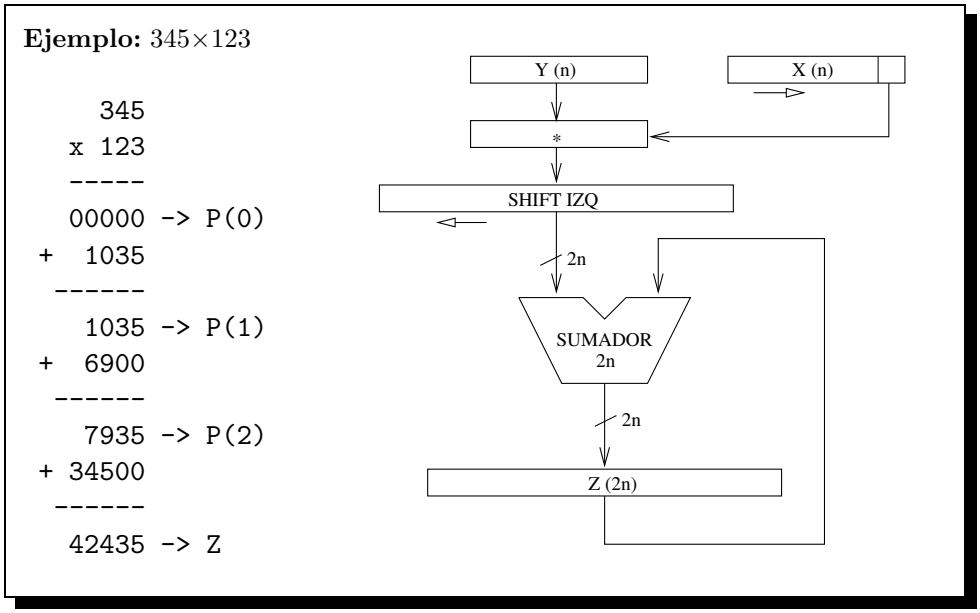


Figura 5.8: Mejora del algoritmo y su implementación

```
P(0)=0;
for (i=0; i<n; i++)
 {
        P(i+1) = x_i × Y × B^i + P(i);
 }
Z = P(n);
```

donde los índices entre paréntesis muestran el mismo elemento en distintos pasos, mientras que los subíndices indican componentes de un elemento.

La realización hardware del algoritmo requeriría, como puede observarse en la figura 5.8, un sumador *Base B* de **2n** dígitos, un multiplicador *Base B* para multiplicar un dígito de X por Y y un desplazador a la izquierda de **2n** dígitos para implementar los desplazamientos asociados al término B^i. Al comienzo del algoritmo el registro Z debe inicializarse a 0. En la primera iteración, *i=0*, el desplazador no opera, en la iteración *i=1*, realizará un desplazamiento a la izquierda, para *i=2*, dos desplazamientos, etc. También, al final de cada iteración del algoritmo, el contenido del registro X debe desplazarse una posición a la derecha, ya que el dígito menos significativo de X es el que multiplica a

Y.

Sin embargo, el coste HW asociado a este algoritmo puede reducirse significativamente. Si se observa el algoritmo, el núcleo del ciclo `for` está constituido por la siguiente expresión suma,

$$P(i+1) = x_i \times Y \times B^i + P(i)$$

donde se aprecian dos sumados, el valor acumulado hasta ese paso, $P(i)$, y la aportación del producto parcial, $PP_i = x_i \cdot Y \cdot B^i$.

Puesto que el producto parcial PP_i está escalado en B^i, la suma no afectará a las componentes *i-1, i-2,..., 1, 0* del valor acumulado en $P(i)$. Además, dados los valores posibles de los dígitos y números de **n** dígitos en *Base B*, se verificará que $x_i \cdot Y < B^{n+1}$ y, por lo tanto, será un número *Base B* de **n+1** dígitos como máximo. Es decir, si el recorrido del índice se realiza de forma creciente, las componentes $B^{2n-1}, B^{2n-2}, ..., B^{n+i+1}$ del valor acumulado $P(i)$ serán nulas. Por todo ello, en cada paso de la iteración, sólo **n+1** dígitos del valor acumulado $P(i)$ serán susceptibles de cambio. En resumen, bastará con un sumador *Base B* de **n+1** dígitos.

En la figura 5.9 mostramos la realización de la multiplicación anterior siguiendo el nuevo algoritmo mejorado de un sumador de **n+1** dígitos.

Ejemplo: 345 × 123

Y
345

	AC	**X**
	0000	12**3**
+	1035	
→	1035	123
	0103	51**2**
+	0690	
→	0793	512
	0079	35**1**
+	0345	
→	0424	351
Z=	**0042**	**435**

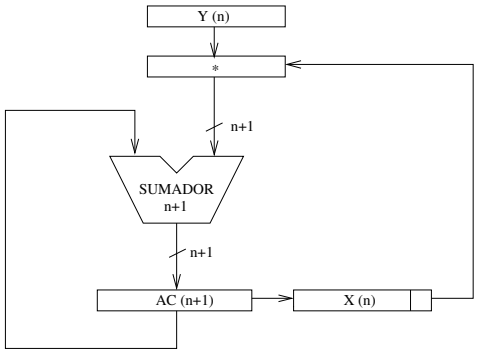


Figura 5.9: Implementación de la segunda mejora del algoritmo

Este algoritmo consiste en inicializar un registro acumulador, **AC**, a cero y luego iterar **n** veces los dos pasos siguientes:

- Sumar con **AC** el dígito menos significativo del registro X por el contenido del registro Y, es decir: $AC \leftarrow AC + (Y \times x_0)$. El dígito x_0 se ha marcado en negrita en el ejemplo.
- Desplazar el par de registros **AC**–**X** una posición a la derecha.

De esta forma, el HW necesario para implementar este algoritmo, también presentado en la figura 5.9, sólo necesita un sumador de **n+1** bits.

5.3.2. Caso Binario. Algoritmo de Suma y Desplazamiento

La particularización del algoritmo del anterior apartado y sus estructuras asociadas, en el caso binario, traen consigo algunas simplificaciones.

Puede observarse que en el caso binario, el cálculo de $x_i \cdot Y$ se reduce a 00...0 si $x_i = 0$, e Y si $x_i = 1$. Además, el producto de $x_i \cdot Y$ es un número de **n** bits, lo que permite utilizar un sumador binario de sólo **n** bits. Deberemos tener en cuenta que al sumar dos números en binario natural de **n** bits el resultado puede ser de **n+1**. Afortunadamente, en el caso de números binarios natural, este bit adicional coincide con el acarreo de salida del sumador. Por ello, en las sumas intermedias, tendremos que considerar este acarreo y salvarlo en un biestable, **F**, para introducirlo en el registro **AC** cuando realicemos el desplazamiento a la derecha. Todo esto se muestra en la figura 5.10.

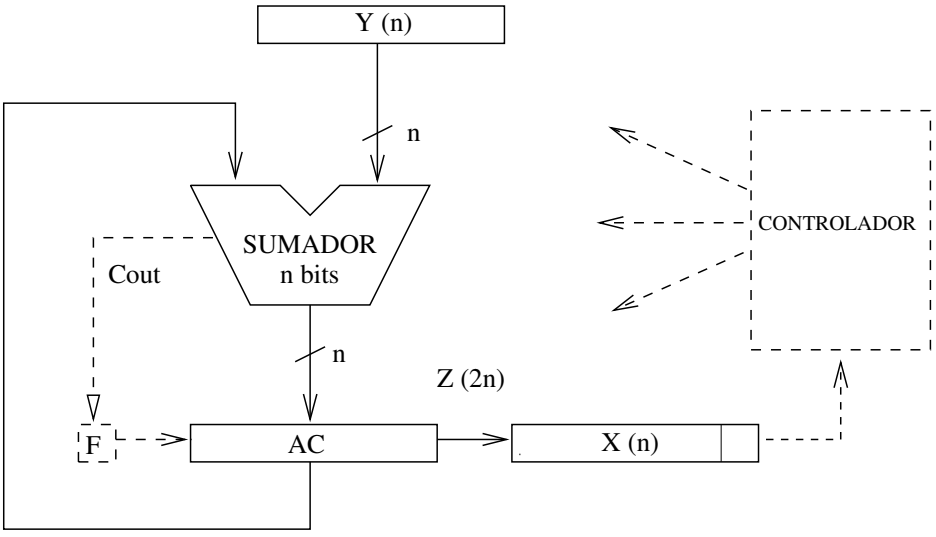


Figura 5.10: Implementación del algoritmo de multiplicación binario

Como vemos, la realimentación del *carry* de salida, C_{out}, a través del biesta-

ble, está pintado con línea discontinua. Esa realimentación puede ser suprimida en el caso de que los números a multiplicar sean positivos y estén representados en *signo/magnitud, C1* ó *C2*, ya que en ese caso serán de la forma `0XXXX...XX`, y en las sumas implicadas en el algoritmo nunca se producirá carry de salida.

Para terminar, mostramos como queda el algoritmo de multiplicación que debe implementar el *Controlador* de la figura 5.10:

```
AC=0;
for (i=0; i<n; i++)
 {
   if (x(0)==1) SUMA Y;
   DESPLAZA;
 }
```

Este algoritmo implementado en el controlador, ya sea mediante control cableado o microprogramado, será el encargado de generar las señales de control apropiadas para que se realicen las sumas y los desplazamientos necesarios, en función del contenido del registro X, cuyo bit menos significativo es testeado desde el controlador.

5.3.3. Mejora para saltar sobre ceros y unos consecutivos (bit-scanning)

En la formulación del algoritmo de multiplicación del apartado anterior, la aparición de 0's en el multiplicador hace que no se realice la suma y sólo se efectué el desplazamiento (saltar sobre 0's). A continuación vamos a demostrar como con un sumador-restador se pueden efectuar tanto saltos sobre cadenas de "0" como de "1" consecutivos. Sea un número binario con una cadena de **k** unos consecutivos a partir de la posición **i**.

```
      <-  k  ->
    j          i
...011......1110....
```

El valor con que contribuye dicha cadena de 1's al número representado será:

$$\begin{aligned}
&\ldots + 2^{i+k-1} + 2^{i+k-2} + \ldots + 2^{i+1} + 2^i + \cdots = \\
&= \ldots + (2^{k-1} + 2^{k-2} + .. + 2^1 + 2^0) \cdot 2^i + \cdots = \\
&= \ldots + (2^k - 1) \cdot 2^i + \cdots = \\
&= \ldots + 2^{i+k} - 2^i + \ldots
\end{aligned}$$

Por lo tanto, la contribución de las **k** sumas puede sustituirse por una suma y una resta. De esta forma, podemos reescribir el algoritmo de multiplicación binaria, ya que si Y es el multiplicando y el multiplicador, X, contiene en su representación una cadena de unos de longitud **k** a partir del bit **i-ésimo**, la ecuación parcial resultante de desarrollar $X \times Y$ queda:

$$Y \cdot (2^{i+k-1} + 2^{i+k-2} + \ldots + 2^{i+1} + 2^i) = Y \cdot 2^{i+k} - Y \cdot 2^i$$

Es decir, el efecto de una cadena de 1's consecutivos en el multiplicador puede sustituirse por restar el multiplicando en la posición en que comienza la cadena y sumar el multiplicando en la posición posterior al último uno de la cadena. Los desplazamientos, en cada paso, se siguen manteniendo.

En pocas palabras, sólo habrá suma o resta del multiplicando al valor acumulado cuando en el recorrido de los dígitos del multiplicador se produce un cambio con respecto al dígito anterior.

Esta idea puede ser plasmada en el código del algoritmo de dos formas distintas:

(a) Con ayuda de un *flag* que informe si estamos o no en una cadena de 1's.
(b) Analizando conjuntamente el dígito del multiplicador correspondiente a la etapa actual y el anterior.

En la opción (b), en lugar de un *flag* se utiliza el bit anterior del multiplicador (al comienzo $x_{-1} = 0$). La decisión sobre si hay que sumar o restar el multiplicando se efectúa según la tabla 5.2, analizando en cada paso de la iteración **i** el bit **i-ésimo** del multiplicador y el anterior, x_{i-1}.

Dado que se realizan sumas y restas, los resultados parciales pueden ser negativos, por lo que habrán de ser almacenarlos considerando su signo. Si en la implementación hardware se utiliza únicamente un sumador binario y la resta se realiza mediante la suma del complemento a dos del substraendo, los resultados negativos saldrán en *C2*. Para respetar el signo de los resultados parciales, los desplazamientos deberán ser aritméticos (extensión de signo). Como siempre, el resultado de sumar o restar dos números de **n** bits puede requerir **n+1** bits, pero en este caso (suma/resta en *C2*) el bit adicional no

x_i	x_{i-1}	**Operación**
0	0	desplazar (cadena de 0's)
0	1	sumar Y y desplazar (fin cadena de 1's)
1	0	restar Y y desplazar (comienzo cadena de 1's)
1	1	desplazar (cadena de 1's)

Tabla 5.2: Operación a realizar en el algoritmo *bit-scanning* según los bits del multiplicador, X

coincidirá con el carry, por lo que habrá que utilizar un sumador de **n+1** dígitos. El algoritmo quedaría:

```
AC=0; X(-1)=0
for (i=0; i<n; i++)
 {
   if (x(i)==0 AND x(i-1)==1) SUMA Y;
   if (x(i)==1 AND x(i-1)==0) RESTA Y; /*+C2(Y)*/
   DESPLAZA;
 }
/* Corrección */
if (UltimaOperacion==RESTA) SUMA Y;
```

donde la última línea del código tiene en cuenta el caso en que el multiplicador (en binario natural) tenga su bit más significativo a uno, con lo cual la última operación sería una resta y el resultado sería incorrecto. Eso se corrige sumando Y después de la última iteración si la última operación fue una resta.

Por último, presentamos un ejemplo de ejecución de dicho algoritmo y el HW necesario para su implementación en la figura 5.11

En general, el número de sumas/restas efectuadas en este algoritmo será menor que el de sumas en el clásico basado en suma y desplazamiento. Sin embargo, la eficiencia depende de los datos y, en el peor caso (secuencia de unos y ceros alternados, ...010101...), requiere más operaciones y, por lo tanto, más tiempo de cómputo que el tradicional de suma y desplazamiento.

5.3.4. Recodificación del Multiplicador

Un modo de reducir el número de iteraciones del algoritmo de suma y desplazamiento es recodificando el multiplicador como un número *Base 2c*, agrupándolo

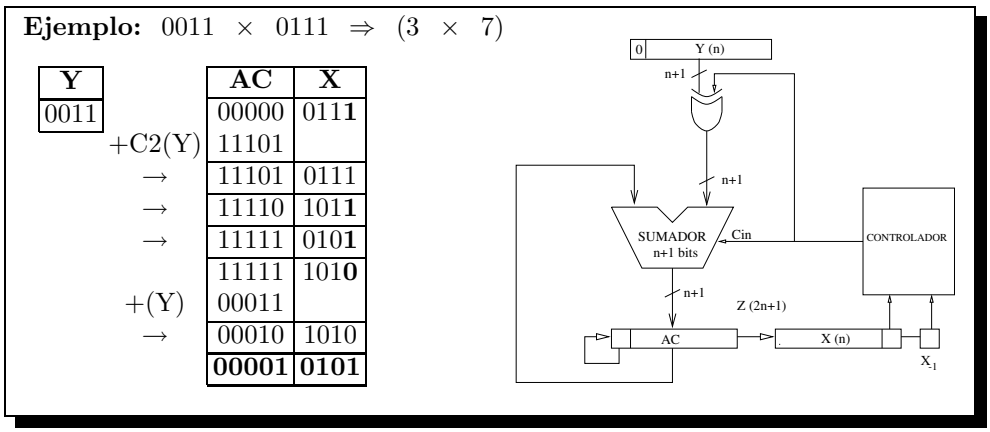


Figura 5.11: Implementación del algoritmo *bit-scanning*

según bloques de **c-bits**. Habrá que disponer de **2c-1** múltiplos del multiplicando. El número de iteraciones se reducirá a $\frac{\mathbf{n}}{\mathbf{c}}$. En cada paso se analizarán **c-bits** y se acumulará el múltiplo correspondiente. Los desplazamientos serán de **c-bits**. En la tabla adjunta mostramos los múltiplos para el caso *c=2*.

x_{i+1}	x_i	**Operación**
0	0	desplazar dos bits
0	1	sumar Y y desplazar dos bits
1	0	sumar $2Y$ y desplazar dos bits
1	1	sumar $3Y$ y desplazar dos bits

Tabla 5.3: Recodificación de los bits del multiplicador (para $c = 2$)

Por ejemplo, si Y=3 y X=30, el multiplicador puede expresarse como:

$$X = 01\,11\,10 = (01)2^4 + (11)2^2 + (10)2^0$$

de forma que:

$$Y \times X = Y \times \left[(01)2^4 + (11)2^2 + (10)2^0\right] = Y \cdot 2^4 + 3Y \cdot 2^2 + 2Y \cdot 2^0$$

Este método, a su vez, puede combinarse con el de salto sobre ceros y unos. En el caso de $c = 2$, esto nos ahorra el cálculo de $3 \cdot X$, que es el de mayor complejidad, ya que implica un desplazamiento y una suma. En el caso en que $c = 1$, el algoritmo se llama de *Booth* cuando los operandos están

representados en *C2*. El algoritmo es exactamente igual al del *bit-scanning* de la página 158 quitando la última línea (la corrección final). Por tanto, el algoritmo de *Booth* permite multiplicar números en *C2* (ya sean positivos o negativos). Al igual que le ocurría al algoritmo *bit-scanning*, un uno (o un cero) aislado provoca una resta y una suma, cuando el algoritmo de suma y desplazamiento sólo da lugar a una suma. De esta forma, el algoritmo de *Booth* genera, en el caso de un multiplicador con unos y ceros alternados, $n/2$ sumas y $n/2$ restas, frente a las $n/2$ sumas que generaría el algoritmo de suma y desplazamiento. Sin embargo, ese inconveniente se resuelve mediante una optimización del algoritmo de *Booth*: el algoritmo de *Booth modificado*, que detecta unos y ceros aislados, por lo que es uno de los más usados.

5.4. DIVISIÓN DE ENTEROS SIN SIGNO

En nuestro estudio de la división sólo nos detendremos en el caso de que los operandos sean enteros sin signo. Muchas máquinas realizan una conversión, desde la representación en *C2* o *C1*, de los operandos involucrados en la división, a su representación en signo magnitud. De esta forma, la división puede realizarse conforme a los algoritmos que se exponen en este apartado, tratando el signo aparte.

5.4.1. División Con Restauración

Cuando realizamos la división entera $Y \div X$, siendo Y el dividendo y X el divisor, buscamos el cociente Q y el resto R. Cuando dividimos dos números en un papel, por ejemplo 13 entre 5, determinamos que el cociente "cabe" a 2 y que el resto es 3 (ya que $13 - (2 \cdot 5) = 3$). La elección que hacemos del cociente, en este caso 2, es fruto de una aproximación que hacemos más o menos mecánicamente, pero que lleva implícita la condición matemática de que el resto (residuo) tiene que ser positivo y menor que el divisor.

Esa aproximación al dígito del cociente apropiado, que nosotros elegimos rápidamente, algorítmicamente sólo se puede implementar mediante el método de ensayo y error. Por ejemplo, en el ejemplo anterior, un algoritmo iría probando cocientes desde el 9, 8, ..., hasta llegar al 2. Es decir, primero probaría el cociente 9, restaría 45 y al resultar un resto negativo (-32), rectificaría volviendo a sumar 45 y cambiando el cociente a 8. Este cociente también da lugar a un resto negativo, así que bajaríamos a 7 y así sucesivamente hasta llegar al cociente 2, que es el primero que da un cociente positivo y menor que el divisor, X.

En el caso binario podemos hacer un análisis parecido, con la simplificación de que las únicas posibilidades de los bits del cociente son 0 y 1. Es decir, se empieza "apostando" por que el bit del cociente será uno, es decir, restando el divisor. En caso de que el resto sea positivo, hicimos una buena apuesta, pero en caso contrario, hay que restaurar y cambiar el cociente a 0. En la figura 5.12 vemos un ejemplo de aplicación de este algoritmo.

Ejemplo: $15 \div 2 \Rightarrow (1111 \div 0010)$

X
0010

	AC				**Y**			
	0	0	0	1	1	1	1	
+C2(X)	1	1	1	0				Resta
	1	1	1	1				
+X	0	0	1	0				Restaura
←	0	0	0	1	1	1	1	← 0
	0	0	1	1	1	1	**0**	
+C2(X)	1	1	1	0				Resta
←	0	0	0	1	1	1	**0**	← 1
	0	0	1	1	1	**0**	**1**	
+C2(X)	1	1	1	0				Resta
←	0	0	0	1	1	**0**	**1**	← 1
	0	0	1	1	**0**	**1**	**1**	
+C2(X)	1	1	1	0				Resta
←	0	0	0	1	**0**	**1**	**1**	← 1
	0	0	1	**0**	**1**	**1**	**1**	
	R= **001**				Q=**0111**			

Figura 5.12: Ejemplo del algoritmo de división con restauración

El circuito más simple que implemente la división binaria deberá ir ajustando metódicamente la posición del divisor en relación con el dividendo y realizar una sustracción. Si el residuo es cero o positivo, se determina el bit del cociente como 1, el residuo se amplía con otro bit del dividendo, el divisor se vuelve a ajustar y se efectúa otra sustracción. Por otra parte, si el residuo es negativo, se determina el bit del cociente como 0, el dividendo se restaura sumándole de nuevo el divisor y este divisor se ajusta para otra sustracción. El algoritmo quedaría entonces como el de la figura 5.13

En la figura 5.14 se muestra un circuito con el que se podría implementar la

```
AC=0;
for (i=0; i<n; i++)
 {
   RESTA X;
   if (AC(n-1)==1)
    {
      RESTAURA (SUMA X);
      DESPLAZA con 0;
    }
   else DESPLAZA con 1;
 }
```

Figura 5.13: Algoritmo de división con restauración

técnica anteriormente expuesta de división con restauración. En él, un divisor positivo de n bits se carga en el **Registro X** y un dividendo positivo de n bits se carga en el **Registro Y**, al principio de la operación. El **Registro AC**, de **n-1** bits, se pone a 0. Cuando la división se termine, el cociente de n bits estará en el **Registro Y** y el residuo en el **AC**. Las substracciones necesarias se facilitarán si usamos aritmética en *Complemento a 2*.

5.5. ALGORITMOS EN PUNTO FLOTANTE

La aritmética con números en punto flotante, expresados mediante una **mantisa** y un **exponente**, se traduce en un algoritmo que termina realizando operaciones enteras con la mantisa y el exponente. Estos algoritmos pueden ser emulados por SW, repercutiendo en un incremento del coste, o se pueden implementar mediante HW. En particular, los coprocesadores aritméticos contienen una gran cantidad de HW dedicada a realizar estos algoritmos para números en punto flotante, permitiendo liberar a la CPU del computador de esa complicada tarea.

5.5.1. Suma y Resta

Dados dos números en punto flotante, $X = M_1 \cdot B^{E_1}$ e $Y = M_2 \cdot B^{E_2}$, su suma se realiza en cuatro pasos (la resta es análoga):

❶ Detectar el operando de mayor exponente.

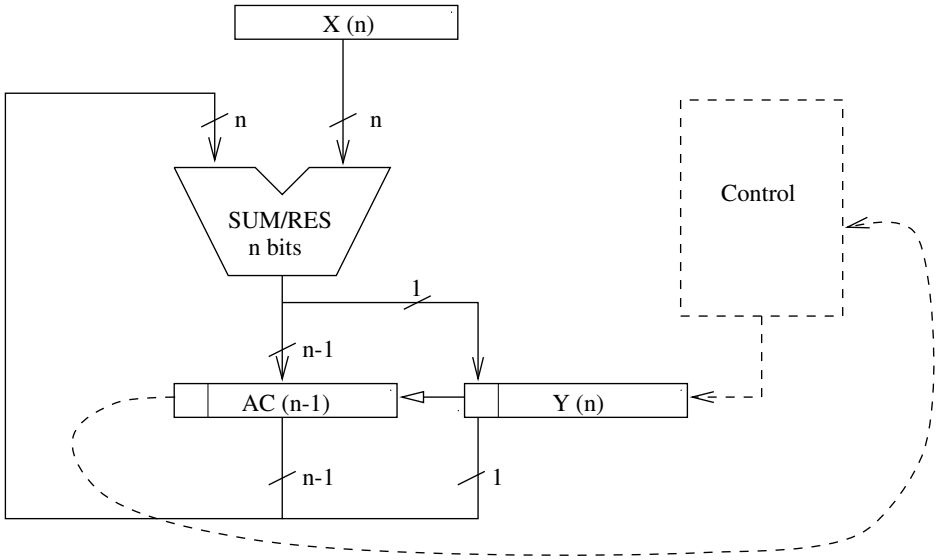


Figura 5.14: Implementación del algoritmo de división binario

❷ Desplazar la mantisa del operando de menor exponente $|E_1 - E_2| \cdot \log_2 B$ posiciones. La base B debe ser una potencia de 2 (B=2 en IE^3–754, B=16 en el IBM-360), para aplicar esta fórmula.
❸ Sumar las mantisas (o restarlas en caso de que tengan distinto bit de signo).
❹ Normalizar.

5.5.2. Multiplicación

Dados $X = M_1 \cdot B^{E_1}$ e $Y = M_2 \cdot B^{E_2}$, su producto P viene dado por:
$P = M_1 \cdot M_2 \cdot B^{E_1+E_2}$
El algoritmo para X e Y normalizados es:

❶ Sumar los dos exponentes, realizando los ajustes necesarios para obtener la notación en exceso correcta.
❷ Multiplicar las mantisas.
❸ Normalizar la mantisa resultante.

Debemos tener en cuenta que si las mantisas ocupan n bits, el producto resultante será de $2n$ bits, por tanto, debemos descartar los n bits menos significativos.

5.5.3. División

Dados $X = M_1 \cdot B^{E_1}$ e $Y = M_2 \cdot B^{E_2}$, su cociente Q viene dado por:

$Q = M_1/M_2 \cdot B^{E_1 - E_2}$

El algoritmo para X e Y normalizados es:

❶ Restar los dos exponentes, realizando los ajustes necesarios para obtener la notación en exceso correcta.

❷ Dividir las mantisas, guardando sólo el cociente.

❸ Normalizar la mantisa resultante.

Si E_1 y E_2 están representados en exceso, el exponente resultante vendrá dado por su diferencia sumándole el exceso. Hay que preguntar si el divisor es cero, para evitar que se produzca una división por cero. Se puede realizar la resta de los exponentes en paralelo con la división de las mantisas.

Si en lugar de utilizar un algoritmo de división (con o sin restauración) se quiere usar un multiplicador en punto flotante, se puede seguir el llamado método de la *división convergente*.

SINOPSIS

Ya se dijo con anterioridad que las instrucciones más frecuentes en los programas son las que deben ser implementadas con más "esmero" para que consuman pocos ciclos de reloj. Pues bien, las instrucciones aritméticas pertenecen a ese tipo de instrucciones que aparecen frecuentemente. Por tanto, no nos debe sorprender que se investigue intensamente en ese campo con la intención de encontrar cada vez mejores técnicas para implementarlas. En este tema sólo hemos introducido algunos principios, pero nos dan una idea de cómo está construido, más o menos, una unidad aritmético lógica o un coprocesador aritmético para números en punto flotante.

RELACIÓN DE PROBLEMAS

1. Realizar las siguientes sumas con números de 6 bits: 12+9, 27-15, 14-19, -7-13, 23+10, -20-13; representando los números mediante los siguientes convenios:
   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2
2. Haz, con números de 8 bits trabajando en
   a) Complemento a 1
   b) Complemento a 2

   las operaciones siguientes:
   - 00101101 + 01101111
   - 11111111 + 11111111
   - 00000000 − 11111111
   - 11110111 − 11110111
3. Encuentra la razón por la que, en el algoritmo de suma/resta en *C1*, hay que sumar el carry de salida al resultado.
4. Los números decimales con signo de $n-1$ dígitos se pueden representar mediante n dígitos sin signo utilizando la representación en complemento a 9. El complemento a 9 es el complemento a la base menos 1 cuando la base es 10 (igual que el C1 es el complemento a la base menos uno cuando la base es 2). El C9 de un número decimal N se obtiene mediante la siguiente expresión:

   $$C9(N) = 10^n - 1 - N$$

   donde n es el número de dígitos con que trabajo. Una técnica para hacer el C9 de un número consiste en restar cada dígito de 9. Así, el negativo de 014725 es 985274. Se pide:
   a) Expresa como números de tres dígitos en complemento a 9 los siguientes números: 6, -2, 99, -12, -1, 0.
   b) Determinar la regla por la cual se suman los números en complemento a 9.
   c) Realizar las siguientes sumas con dicha técnica:
      1) 0001 + 9999
      2) 0001 + 9998
      3) 9997 + 9996
      4) 9241 + 0802

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

5. Los números en complemento a 10 son análogos a los números en complemento a 2. Un número negativo en C10 se forma con sólo sumar 1 al número correspondiente en C9, prescindiendo del acarreo. Más formalmente, el C10 de un número decimal N se obtiene mediante la siguiente expresión:
$$C10(N) = 10^n - N$$
¿Cuál será la regla para la adición en complemento a 10?
6. Cuando realizamos operaciones de suma/resta en decimal, realmente estamos utilizando un algoritmo de suma/resta para números decimales representados en signo/magnitud, donde el "dígito" de signo esta representado por el + o el − que antecede al número. Realiza, con aritmética decimal de tres dígitos más el signo, las siguientes sumas y comprueba que el algoritmo que debes utilizar es semejante al visto en teoría para los números en binario: $(-264)+(-858)$, $(+858)+(-264)$, $(+264)+(-858)$.
7. Árbol de Wallace
   a) Dibujar el árbol de Wallace para sumar 6 números enteros mediante CSA's (Carry Save Adders). ¿Cómo ha de ser el último sumador?
   b) Sumar $12 + 5 + 7 + 3 + 10 + 9$ siguiendo la estructura previa.
8. ¿Cuándo tiene sentido utilizar sumadores con acarreo almacenado (CSA)? Construir el Árbol de Wallace para sumar $7 - 5 + 9$ y realizar la suma utilizando aritmética de 6 bits en C2.
9. Realiza las siguientes multiplicaciones mediante el algoritmo de multiplicación binario de suma y desplazamiento con n=5.
   a) 12×10
   b) 7×12
   c) 6×15
10. Realiza las siguientes divisiones mediante el algoritmo de división binario con restauración con n=5.
   a) $12 \div 3$
   b) $15 \div 2$
11. Realiza las siguientes operaciones con números en punto flotante en el formato IEEE 754.
   a) $C30C0000 + C1500000$
   b) $3B370000 + 39F68000$

6 Memorias

OBJETIVOS

- Entender los principios de localidad y jerarquía de memoria
- Introducir las técnicas de memoria entrelazada, asociativa y caché

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

6.1. ANCHO DE BANDA, LOCALIDAD Y JERARQUíA DE MEMORIA

La **Memoria** contiene la mayor parte de las instrucciones y datos del programa que se está ejecutando. Lo que no cabe se almacenará en memoria secundaria, por lo que, si consideramos ésta como parte de la memoria, podemos afirmar que toda la información de nuestros programas se encuentra en memoria.

Las dos características más importantes de la memoria son el tamaño y el ancho de banda. El **tamaño** es su capacidad, es decir, el número de palabras que puede almacenar. El **ancho de banda** es el número de palabras que podemos acceder por unidad de tiempo, determinando por tanto la velocidad a la que podemos trabajar con ella.

La disparidad entre la velocidad a la que son capaces de computar hoy día los procesadores y el ancho de banda de las memorias plantea un importante problema. Esto es debido a que el micro procesa mucho más rápido que la velocidad a la que las memorias son capaces de suministrarle los datos. La situación ideal se alcanzaría si *Velocidad del Procesador = Velocidad de la Memoria*.

Para paliar este problema se alcanza un compromiso. Se invierte un dinero en tecnología de alta velocidad para aumentar el ancho de banda de la memoria. Lo ideal sería construir una memoria suficientemente grande y con gran ancho de banda, pero la tecnología es tan cara que resulta prohibitivo, por lo que sólo se usará cuando el gasto adicional que supone justifique la inversión. Tendremos, por lo tanto, un compromiso entre tamaño y ancho de banda, de modo que si aumentamos uno se debe reducir otro si queremos mantener un precio constante.

La solución que se adopta, siguiendo siempre criterios económicos y de factibilidad, es usar diferentes configuraciones de memoria más rápidas (y pequeñas) donde podamos sacar alguna ventaja por la inversión adicional que suponen. Las estrategias más usadas son:

- Entrelazamiento de memoria
- Empleo de memorias asociativas (se usan con memorias caché y virtual)
- Memoria caché
- Memoria virtual

La efectividad de estas soluciones se basan en el *principio de localidad*, que puede ser formulado según dos vertientes:

a) Localidad Temporal: Si estamos referenciando la posición de memoria I, es muy probable que la volvamos a referenciar en un breve periodo

de tiempo, es decir, que la información que utilizo en un momento dado la volveré a utilizar pronto. Se puede ver claramente que esto es lo que sucede en los lazos.

b) Localidad Espacial: Si la posición de memoria I es referenciada, es muy probable que a continuación se referencien direcciones cercanas a I o contiguas en memoria. Esto es lo que ocurre cuando accedemos a datos relacionados como arrays, matrices. Se debe a que los programas, en su mayor parte, son secuenciales.

El principio de localidad puede resumirse en lo que se denomina la *Regla 90/10*, que nos dice que el 90 % del tiempo de ejecución de un programa se consume en el 10 % del código. Si estos dos principios no se cumpliesen, no tendría sentido aplicar las técnicas de memoria que vamos a estudiar en este tema.

Jerarquía de Memoria

La aplicación de las técnicas de memoria caché, virtual, entrelazada... se lleva a cabo implementando una jerarquía de memoria de la siguiente forma:

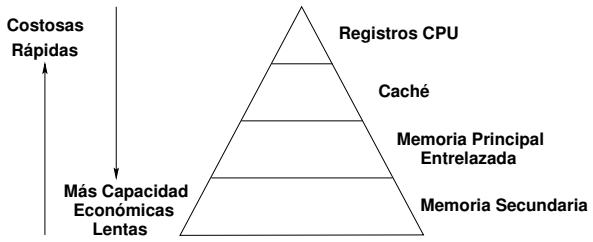


Figura 6.1: Jerarquía de Memorias

En la parte más alta de la pirámide se sitúan las memorias más rápidas, que al ser también las más costosas serán las de menor tamaño. En el otro extremo, tendremos las memorias de más capacidad, que además serán las más lentas y económicas.

6.2. ORGANIZACIÓN DE LA MEMORIA

6.2.1. Memoria Entrelazada

Esta técnica consiste en dividir la memoria en módulos a los que podemos acceder de manera independiente, de modo que, en un mismo instante, es posible

trabajar con todos los módulos a la vez. De este modo podemos multiplicar el ancho de banda (en el mejor de los casos) por el número de módulos que estemos utilizando.

Vamos a suponer que tenemos una memoria de $N = 2^n$ palabras y que ésta se encuentra repartida en $M = 2^m$ módulos. Hay dos posibles esquemas, cada uno con sus ventajas e inconvenientes:

a) Orden Superior: Palabras consecutivas van a parar al mismo módulo.

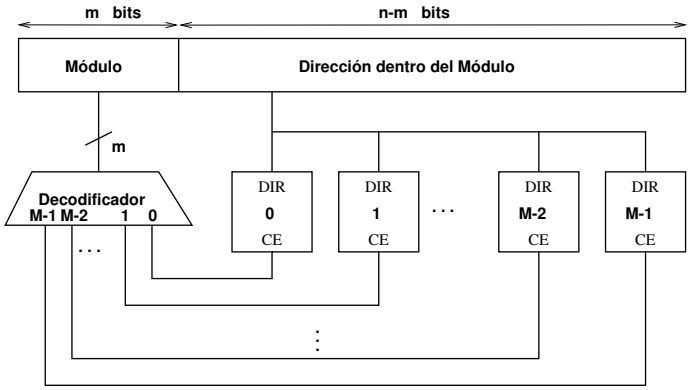


Figura 6.2: Memoria Entrelazada de Orden Superior

b) Orden Inferior: Palabras consecutivas van a parar a módulos consecutivos.

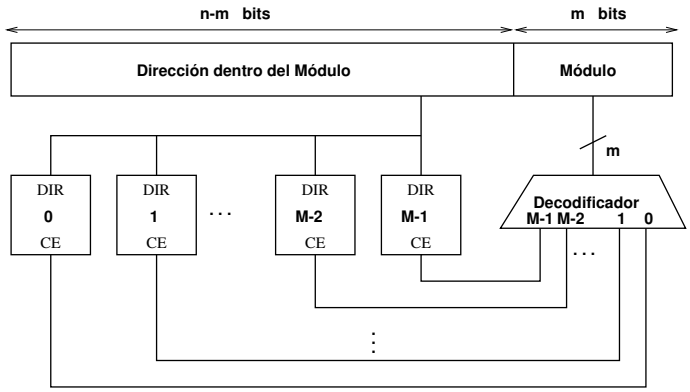


Figura 6.3: Memoria Entrelazada de Orden Inferior

El rendimiento de este tipo de memorias se degrada cuando tenemos con-

flictos de memoria. Un *conflicto de memoria* se produce cuando dos accesos consecutivos se refieren al mismo módulo. En el peor de los casos, cuando siempre direccionamos al mismo módulo, no tendremos ninguna ganancia con respecto a una memoria convencional (un solo módulo). Sin embargo, en el mejor de los casos, si conseguimos que M direcciones consecutivas se refieran a cada uno de los M módulos, lograremos multiplicar la velocidad del módulo por M.

Si observamos el funcionamiento de cada una de las posibles configuraciones, veremos como se darán más conflictos en la de orden superior, fenómeno explicado por el principio de localidad espacial.

Sin embargo, las memorias entrelazadas de orden superior tienen una serie de ventajas:

- Se usan en sistemas multiprocesador, en los que cada procesador accede a su propio módulo.
- Se expande con facilidad. Ampliar su tamaño será poco más complejo que añadir nuevos módulos.
- Si se rompe un módulo, solo afectará a un área de memoria localizada.

De cualquier modo, normalmente se usa la técnica de entrelazamiento de orden inferior, ya que es la que minimiza el número de conflictos.

Rendimiento de Memorias Entrelazadas

En este apartado se analizará el rendimiento de un sistema de memoria entrelazada de orden inferior, considerando M el factor de entrelazamiento. Supondremos que el factor limitante es la operación de memoria, despreciando el tiempo de carga de los latches. El cálculo del rendimiento se realiza para una traza equiespaciada con distancia entre las peticiones Q, es decir, la solicitud de peticiones es $\{0, Q, 2Q, 3Q, ...\}$.

- Para el caso de *entrelazamiento con latches a la salida*, tendremos que, si $Q \geq M$, es imposible leer en paralelo más de una petición, ya que la distancia entre ellas es mayor que el rango de posiciones guardada simultáneamente en los latches. Si por el contrario $Q < M$, podemos obtener más de una petición en paralelo con una sóla lectura simultánea de los módulos. En esta situación, si $Q = 1$, tendremos el máximo paralelismo, ya que accedemos a todo el rango de posiciones presentes en los latches tras el acceso a los módulos de memoria. Si $1 < Q < M$, es la relación M/Q la que limita el número de accesos disponibles en los latches tras el acceso a los módulos. La siguiente expresión proporciona

la aceleración (α) del sistema bajo las restricciones mencionadas [11]:

$$\alpha = \begin{cases} 1 & \text{si } Q \geq M \\ \left\lceil \frac{M}{Q} \right\rceil & \text{si } 1 \leq Q < M \end{cases}$$

- En el caso de *entrelazamiento con latches a la entrada*, la limitación en el paralelismo de acceso a los módulos viene dado por las colisiones de acceso a un módulo. Como la traza que analizamos está constituida por peticiones de la forma kQ, con $k \in \mathbb{N}$, existirá colisión (acceso al mismo módulo) entre dos peticiones $k_A Q$, $k_B Q$ si su diferencia es múltiplo de M (ya que dichas peticiones se hayarán mapeadas en el mismo módulo). Por tanto, la aceleración del sistema será aquél valor natural, más pequeño, que multiplicado por Q proporciona un múltiplo de M, es decir:

$$\alpha = min\{k \in \mathbb{N} \mid k \cdot Q = \overset{\bullet}{M}\} = \frac{\text{m.c.m.}(Q, M)}{Q} = \frac{M}{\text{m.c.d.}(Q, M)}$$

6.2.2. Memoria Asociativa

Consideremos la siguiente tabla, una estructura bastante usada para organizar la información:

APELLIDO	PESO	TALLA	EDAD
PÉREZ	63	1.71	27
GARCÍA	90	1.87	45
LÓPEZ	82	1.83	33
MARTÍN	51	1.64	61
MILLER	100	2.05	24
ANSLEY	110	2.04	25

La información en la tabla está formada por una serie de entradas llamadas *registros* (uno por cada línea), y estos a su vez se dividen en *campos* (en el ejemplo son peso, talla y edad).

A la hora de acceder (leer o escribir) a la tabla tenemos varias posibilidades.

a) La más común es la que proporcionan las memorias con acceso aleatorio, es decir, especificando la dirección física del dato que queremos acceder. La dirección física no tiene ninguna relación lógica con la información

[11] Con $\lceil x \rceil$ indicamos el *ceil* de un número, es decir el entero más cercano hacia $+\infty$, por ejemplo $\lceil 3.5 \rceil = 4$, y $\lceil -3.5 \rceil = -3$. El entero más cercano hacia $-\infty$ se denota $\lfloor x \rfloor$ y se conoce como función *floor*.

que almacena, por lo que este método introduce cierta complejidad lógica al ser un método de acceso artificial.

b) Emplear uno (o varios) campo(s) para direccionar la información que queremos localizar. Por ejemplo, nos podría interesar conocer los datos correspondientes a la persona cuyo campo $TALLA = 1.83$. En este tipo de memoria usaríamos el campo $TALLA$ para direccionar y ésta nos devolvería el registro completo: LÓPEZ, 82, 1.83, 33.

Esto también se podría conseguir con una memoria normal, pero para ello habría que recorrer toda la tabla secuencialmente en busca de las posibles concordancias del campo deseado. Al tener que acceder secuencialmente a todas las posiciones de memoria, este método sería muy *lento*. La ventaja de las memorias asociativas es que las búsquedas de todas las entradas se realiza simultáneamente.

Podemos definir la *memoria asociativa* como aquella que tiene capacidad de acceder a una palabra almacenada usando como dirección un subcampo de dicha palabra. La búsqueda de ésta la realiza en paralelo. Este tipo de memorias recibe otros nombres: MDC o memoria Direccionable por Contenido (CAM en inglés); memoria de búsqueda en paralelo o memoria multiacceso.

Se usa en gran variedad de aplicaciones, como:

- Gestión de memoria caché.
- Gestión de memoria virtual.
- Manipulación de información almacenada en bases de datos.
- Tratamiento de señales de rádar, imágenes.
- Inteligencia artificial, etc.

El hecho de que se utilice solo en aplicaciones muy concretas es debido a que tienen un elevado coste, pues implica redundancia del hardware. Su coste es bastante superior a una memoria RAM convencional.

La estructura del hardware que implementa este tipo de memorias aparece representada en la figura 6.4.

Las Celdas C_i son capaces de almacenar un *registro* completo (apellido, peso, talla, etc,...). La gestión de los accesos los realiza la Unidad de Control (U.C.), que posee dos registros básicos:

- Comparando (C), donde se escribe la información a buscar.
- Máscara (M), que dice qué campos del registro queremos comparar. Los bits no enmascarados del registro `C` será comparados con los bits correspondientes de todas las celdas de la memoria asociativa.

Los registros Indicadores I_i se ponen a 1 si su registro asociado C_i contiene la información que se estaba buscando. El Dispositivo de Evaluación de Datos

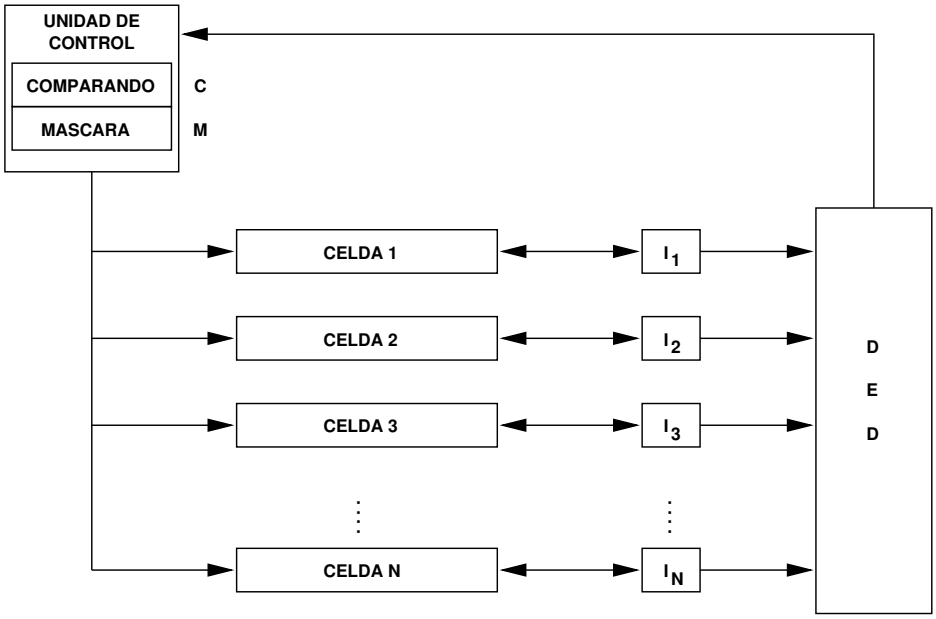


Figura 6.4: Diagrama de Bloques de una Memoria Asociativa

(D.E.D.) es el encargado de informar a la Unidad de Control de las celdas que contienen la información buscada.

El mecanismo de funcionamiento es el siguiente:

❶ La Unidad de Control pone todos los I_i a 1.
❷ Se escribe el registro Comparando.
❸ Se pone el valor que corresponda en el registro de Máscara.
❹ Tras ello, todos los registros indicadores correspondientes a las celdas cuyo valor, para los bits en los que la máscara valga 1, coincida con el comparando, se quedarán con el valor de 1 y el resto se pondrán a 0.

Operaciones de búsqueda típicas en memorias asociativas son: búsquedas de extremos (máximos, mínimos), de equivalencia (igual a, distinto a), por umbral (mayor que, menor que) y selecciones de valores ordenados según un campo (ascendente, descendente).

La principal ventaja de estas memorias estriba en que la complejidad de los algoritmos mencionados es función del tamaño de las celdas y no de cuántas celdas estén ocupadas en la memoria. En otras palabras, buscar un número máximo en una memoria asociativa no depende de cuantos números contenga

la memoria, sino de cuantos bits se utilizan para representar cada número.

6.2.3. Memoria Caché

Es una memora rápida, cara y pequeña. Se encuentra situada entre el procesador y la memoria principal. En principio se puede considerar que funciona como un buffer de la memoria principal.

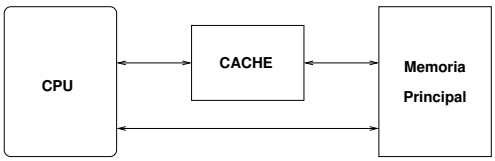


Figura 6.5: Ubicación de la Memoria Caché

La caché se usa para almacenar una copia de aquellas palabras de la memoria principal que están siendo usadas actualmente (y que, según el principio de localidad son las que se usarán con mayor probabilidad en un futuro cercano), ya que, debido a su reducido tamaño, no podemos almacenar toda la información que nuestro programa necesita.

Operación

El procesador, cuando tiene que acceder a memoria, siempre mira antes si la información que busca se encuentra en la memoria caché. Pueden darse dos posibilidades:

- ✔ La información solicitada se encuentra en la caché, *acierto*: se lee/escribe la copia en caché.
- ✘ La información solicitada NO se encuentra en la caché (*fallo*): leo de memoria principal la información buscada y guardo una copia en la caché. A la hora de escribir un nuevo dato en la caché pueden ocurrir dos cosas:

  ❶ Hay espacio libre en la caché: escribo el dato en alguna de las posiciones libres.

  ❷ NO hay espacio libre en la caché: debo buscar qué dato de los que se encuentran en la caché sustituiré por el que acaba de llegar. La elección de las palabras a sustituir se realiza según algún criterio: *RAND* (Aleatoriamente), *FIFO* (la que lleva más tiempo en la caché) o *LRU* (menos usada recientemente). Si algún dato de los que se van a sacar de la caché se modificó por el programa, se deberá actualizar la

copia existente en memoria principal, para mantener la *coherencia de caché*.

La base del éxito del sistema es la localidad en los accesos durante la ejecución de los programas. Podemos mencionar los siguientes aspectos que determinan el rendimiento de la caché:

1. Índice de aciertos o probabilidad de que un dato se encuentre en la caché.
2. Tiempo de acceso a caché.
3. Tiempo de reemplazo.
4. Tiempo consumido en la reparación de las inconsistencias.

Organización

La memoria caché se divide en dos partes, que se representan en la figura 6.6. Consideramos una memoria principal referenciada a nivel de palabra (las direcciones son direcciones de palabra). Denominamos línea a la unidad básica de transferencia entre memoria principal y caché. Cada línea se compone de varias palabras consecutivas de memoria y es la unidad mínima de datos que intercambian caché y memoria principal.

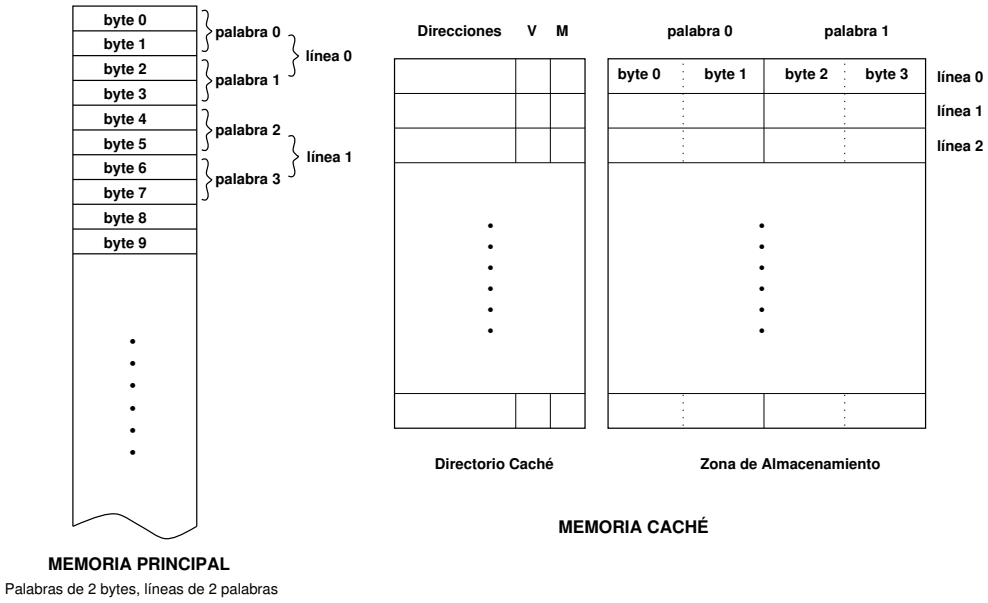


Figura 6.6: Estructura de una Memoria Caché

El *Directorio de la Caché* guarda la dirección de comienzo de la línea que

almacena para esa entrada en la caché. Tiene, además, 2 bits por línea: **V** (Válido, indica si el espacio reservado para almacenar la línea contiene información válida) y **M** (Modificación, indica si la línea ha sido modificada). Normalmente, el directorio es una memoria asociativa, de forma que la búsqueda de una palabra en la memoria caché se hace en paralelo.

La *Zona del Almacenamiento* es una memoria convencional, de gran velocidad de acceso, que puede guardar tantas líneas de memoria principal como entradas tiene el directorio.

Veamos a continuación un ejemplo de cómo funciona la memoria caché. Supongamos que en el instante que vamos a tomar como inicial, en la memoria principal se encuentran los datos que aparecen en la Figura 6.7 para las posiciones indicadas. En este ejemplo, vamos a suponer también que las líneas está formadas por 4 bytes.

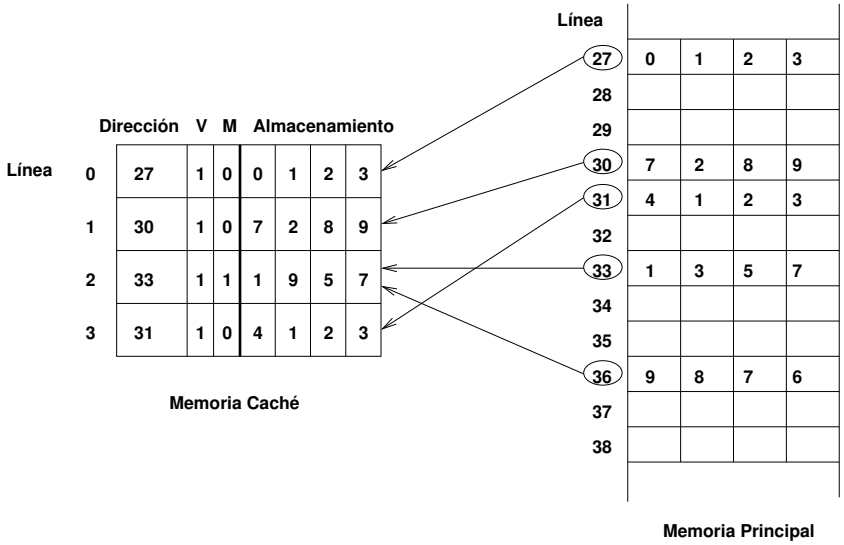


Figura 6.7: Contenidos de la Memoria del Ejemplo

La caché se va a encontrar inicialmente vacía, con datos no válidos, por lo que sus bits V_i van a valer todos 0. A continuación, el procesador realiza una secuencia de accesos a las líneas de memoria siguientes:

27 : Fallo. Almacenar **27** en la Dirección 0, Poner $V_0 = 1$.
30 : Fallo. Almacenar **30** en la Dirección 1, Poner $V_1 = 1$.
33 : Fallo. Almacenar **33** en la Dirección 2, Poner $V_2 = 1$.
31 : Fallo. Almacenar **31** en la Dirección 3, Poner $V_3 = 1$.

33 : Acierto. La línea está en la caché. Hacemos una escritura en el byte 0, escribiendo un 9 en él, por lo que el bit M se pondrá a 1 (línea modificada).

30 : Acierto.

31 : Acierto.

27 : Acierto.

36 : Fallo. Habrá que traer su valor de memoria principal a caché. Como la caché se encuentra ya llena, tendremos que reemplazar alguna línea. La línea elegida dependerá del algoritmo de reemplazo usado:

* FIFO → se reemplaza la más antigua, la **27**.
* LRU → se sustituye la que lleva más tiempo sin referenciar, que en nuestro caso es la línea **33**.
* Aleatorio → se escoge cualquiera de ellas al azar.

En caso de que haya que eliminar la línea **33** de la caché, como se modificó, antes de borrarla y escribir en su sitio la **36**, hay que actualizar su copia (no válida ya) que existe en memoria principal.

En la figura 6.7 podemos ver el estado de la memoria caché, justo después de que el procesador escriba en la línea **33**.

Visto el funcionamiento básico de una caché, veamos diferentes organizaciones que pueden existir en la **zona de almacenamiento**. El problema que queremos resolver es cómo mapear los datos de la memoria principal en la memoria caché, existiendo para ello diferentes criterios:

- Asignación directa: líneas consecutivas en memoria principal van a parar a líneas consecutivas en la memoria caché.
- Asignación completamente asociativa: las líneas de memoria principal pueden ubicarse en cualquier línea de la memoria caché.
- Asignación asociativa por conjuntos: cuando se usa esta estrategia, las líneas de memoria caché se agrupan en conjuntos del mismo tamaño. A cada línea de memoria principal se le asigna un conjunto, pero dentro de éste puede ocupar cualquier posición.

Como conclusión sí podremos decir que si el número de fallos de caché es pequeño, la CPU casi esta "viendo" una memoria tan grande como la memoria principal y tan rápida como la memoria caché. Todo esto a un precio mucho más razonable que si diseñamos el sistema computador con una caché tan grande como la memoria principal.

SINOPSIS

La jerarquía de memoria surge como consecuencia de la localidad existente en la ejecución de los programas. En el diseño de esta jerarquía hay muchos parámetros a considerar, pero siempre tendrá un único objetivo: maximizar el ancho de banda y el tamaño y minimizar el coste. El resultado más remarcable es que una jerarquía de memoria bién diseñada crea la ilusión de constituir una memoria única tan grande como la memoria principal (o disco duro si consideramos el nivel de memoria virtual) con el ancho de banda de la memoria caché. Y todo con con un coste mucho más razonable que el de una prohibitiva memoria caché del tamaño de la memoria principal o de un disco duro.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

7 Entrada/Salida

OBJETIVOS

- Recalcar la relevancia de la unidad de Entrada/Salida (E/S)
- Distinguir entre E/S mapeada en memoria o por acumulador
- Introducir los conceptos de E/S programada, por interrupciones y por DMA

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

7.1. INTRODUCCIÓN

En los capítulos anteriores hemos estudiado la CPU y la memoria. Nos queda hablar de la E/S. En este capítulo no vamos a tratar a los periféricos en sí, sino los mecanismos de comunicación con los periféricos.

En la Figura 7.1 encontramos una visión general de un computador, con los distintos bloques que se conectan al bus: CPU, memoria y los controladores de los distintos periféricos del computador (impresora, red, etc).

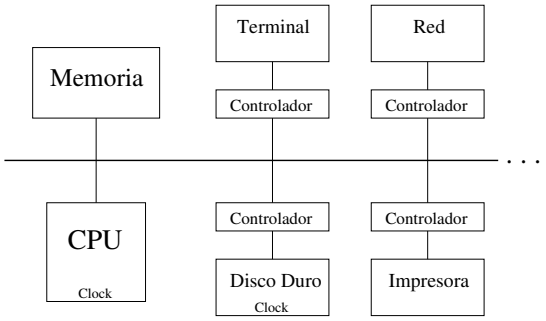


Figura 7.1: Diagrama de bloques de un sistema basado en procesador

Los periféricos no pueden conectarse directamente al bus por tres motivos:

Sincronización : los periféricos tienen un reloj interno de frecuencia y fase diferente a la de la unidad central.

Velocidad : el acceso a periféricos es unos 3 órdenes de magnitud más lento que el acceso a memoria principal.

Codificación : los sistemas de representación de la información en la CPU son diferentes a los usados por los periféricos.

Por estos motivos, es necesario colocar un **controlador** entre cada periférico y el bus. Mediante estos controladores vamos a intentar adecuar la velocidad de los periféricos a la de la CPU.

Insistimos aquí en que un sistema CPU-Memoria no tiene ninguna utilidad si no puede comunicarse con el exterior, es decir, sin un subsistema de E/S. Por tanto, si queremos que el sistema completo sea rápido, no bastará con tener una CPU rápida y un acceso a memoria rápido si es frecuente el acceso a los periféricos.

Por ejemplo, si los accesos a disco duro son lentos, el sistema completo se ralentizará, tanto más, cuanto más frecuentes sean los accesos a disco duro. De nuevo puede aparecer el problema de cuello de botella que manifiesta el

sistema de memoria: el procesador consume los datos a mayor velocidad que la que los periféricos pueden alcanzar para suministrar esos datos.

7.1.1. Organización

La CPU, memoria y periféricos comparten el bus del sistema, tanto el bus de datos como el bus de direcciones. A cada unión (*interfaz*) entre el bus y el periférico se le llama **puerto** (de E/S). Es la "puerta" a través de la cual accedemos a los controladores (y, por tanto, a los periféricos que tienen asociados). Físicamente, un puerto de entrada/salida no es más que un registro que puede ser leído por la CPU (y escrito también en muchos casos).

Para poder seleccionar qué puerto queremos leer (volcar su contenido al bus de datos), asignamos a cada uno de los puertos una dirección única. De esta forma, para leer el contenido de un puerto, primero lo direccionamos (colocando en el bus de direcciones su dirección particular) y luego leemos su contenido.

En la figura 7.2 podemos ver un esquema de este sistema, incluyendo los controladores y puertos. Como se puede observar, podemos tener más puertos que periféricos. En esta figura no hemos incluido las líneas de control, ya que en función del conexionado de este bus de control tendremos dos posibles configuraciones que veremos a continuación.

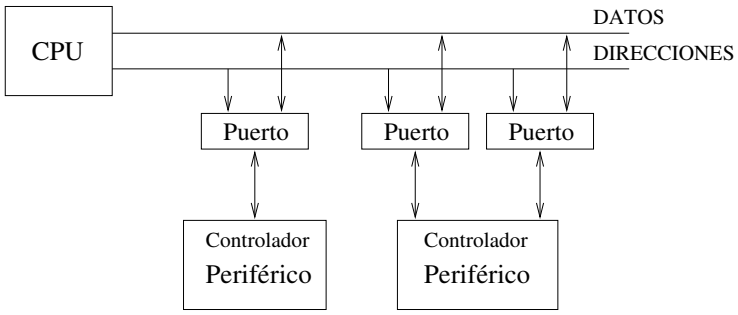


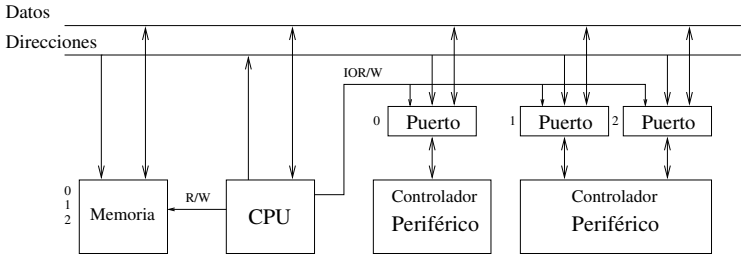
Figura 7.2: Esquema general de un sistema basado en procesador y sus periféricos

a) E/S mapeada en E/S o por acumulador

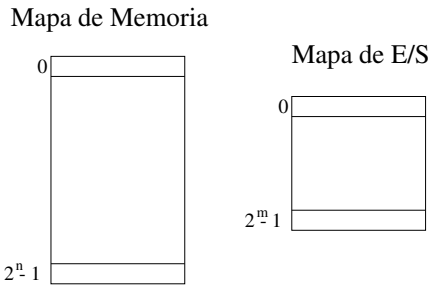
En este caso tendremos diferentes señales de control para la memoria y para los puertos. Es decir, en cierto sentido, el bus de control está dividido: unas

líneas son para el control de los dispositivos de memoria, mientras que otras líneas se dedican al control de los periféricos.

Por ejemplo, en la figura 7.3 vemos como la línea de selección de lectura/escritura en memoria es distinta de la línea de lectura/escritura en alguno de los puertos.



(a) Diagrama de Bloques



(b) Mapa de Direcciones

Figura 7.3: E/S Mapeada en E/S

De esta forma, según la señal de control que se active (R/W ó IOR/W), sabremos a quién hacen referencia las direcciones del bus de direcciones. Esta situación, llevada al nivel del lenguaje ensamblador del procesador, se traduce en que tendremos diferentes instrucciones para transferencia de datos dependiendo de que las transferencias sean entre CPU/memoria o entre CPU/periféricos. Por ejemplo, en los micros de Intel la instrucción `MOV` se utiliza para transferencias con memoria, mientras que las instrucciones `IN` y `OUT` para transferencias con puertos de E/S.

El **mapa de memoria** o **mapa de direcciones** representa el rango de

direcciones válidas que puede colocar la CPU en el bus. Dado que existen direcciones de memoria y direcciones de puertos de E/S, tendremos mapas de memoria independientes para cada uno de los conjuntos de direcciones.

En la E/S por acumulador, al poder distinguirse un acceso a memoria de otro a periférico gracias a que las señales de control son independientes, los *mapas de memoria* pueden solapar parte de su espacio de direcciones. Por ejemplo, en las arquitecturas con procesadores Intel, la parte más baja de los espacios de direcciones pueden corresponder tanto a direcciones de memoria como de E/S.

b) E/S mapeada en memoria

En este caso, la memoria y los puertos tienen las mismas señales de control, es decir, y volviendo al ejemplo anterior, sólo dispondremos de una señal de lectura/escritura (R/W). Por tanto, las instrucciones de transferencia de datos en ensamblador serán las mismas para ambos tipos de transferencia. Por ejemplo, en los procesadores de Motorola, la instrucción `MOVE` se puede utilizar tanto para lectura/escritura de posiciones de memoria como de puertos de E/S.

De esta forma, si queremos distinguir entre un acceso a memoria y un acceso a periférico, debemos asignar rangos de direcciones disjuntos a ambos tipos de dispositivos. Es decir, en un sistema con E/S mapeada en memoria, el espacio de direcciones de E/S no se puede solapar con el espacio de direcciones de posiciones de memoria, como vemos en la figura 7.4.

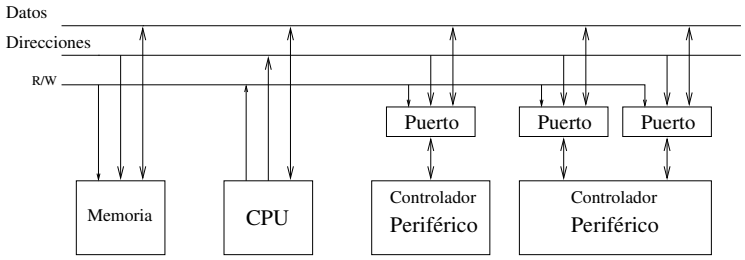
Para terminar esta sección de introducción, vamos a enumerar las distintas estrategias existentes para comunicar una CPU con sus periféricos. Éstas serán estudiadas a continuación en las próximas secciones. Por orden de complejidad ascendente, estas técnicas son:

- E/S dirigida por programa.
- Interrupciones.
- Acceso Directo a memoria (DMA).

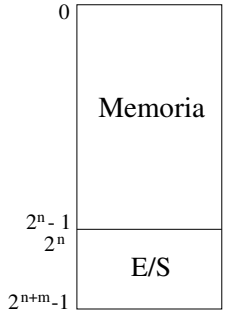
7.2. E/S DIRIGIDA POR PROGRAMA

Esta es la técnica más simple para controlar operaciones de E/S. Al mismo tiempo, y como es de esperar, también es la técnica menos eficiente y lenta para transferencia de información con periféricos.

La idea es que la CPU controle totalmente las operaciones de E/S. La CPU tiene, por tanto, que ejecutar programas para iniciar, dirigir y finalizar



(a) Diagrama de Bloques



(b) Mapa de Direcciones

Figura 7.4: E/S Mapeada en Memoria

las operaciones de E/S. Veámoslo a través del siguiente ejemplo. Supongamos que la CPU está leyendo un fichero del disco duro (HD), y que este dispositivo presenta una controladora con dos puertos de un byte (A y B). El puerto A (con dirección 15) será el puerto de datos, es decir, el puerto que lee la CPU cuando la controladora ha leído un byte del HD y lo ha escrito en ese puerto. Para indicar que el puerto A tiene un byte válido, la controladora pone el bit 0 del puerto B (dirección 16) a uno.

Con esta situación, y partiendo de que inicialmente el bit 0 del puerto B, B(0), está a 0, el proceso de lectura de fichero dirigido por programa consta de las siguientes fases:

1. Seleccionar el puerto B, es decir, colocar en el bus de direcciones la dirección 16.
2. Leer el bit 0 del puerto B, hasta que el bit sea uno. A esta operación se

le llama **espera activa**. Se dice que la espera es activa porque la CPU está ejecutando instrucciones para leer el bit 0 del puerto B hasta que esté a uno.

3. Una vez sabemos que hay un byte válido en el puerto A, lo direccionamos (ponemos en el bus de direcciones la dirección 15) y leemos del bus de datos. Guardaremos el byte en un registro temporal de la CPU.
4. A continuación, realizamos la transferencia entre el registro de la CPU y alguna posición de memoria.
5. Si no hay más transferencias terminamos, y, si no, volvemos al paso 1.

Las principales ventajas de este sistema son su bajo coste hardware y su simplicidad. Presenta el problema de las bajas velocidades conseguidas y de la ralentización del procesador en la espera activa.

7.3. INTERRUPCIONES

Una **interrupción** es un mecanismo que permite a la CPU ser sensible a estímulos diferentes a los creados por sus operaciones internas y, en particular, a los generados por los dispositivos de E/S. Mediante este mecanismo, una vez que la CPU indica a algún dispositivo de E/S qué tarea debe realizar, se desentiende de éste. La CPU puede realizar mientras otras tareas, hasta que el periférico le avise de que ha completado la operación que se le había encargado. Para "avisar" a la CPU se utiliza su señal de petición de interrupción. Como vimos en el tema 3, cuando la señal de interrupción de la CPU se activa, ésta ejecuta una rutina llamada **rutina de tratamiento de interrupción**, en la que la CPU atiende al periférico que la ha interrumpido.

7.3.1. Clasificación de las Interrupciones

Interrupción Hardware : se generan cuando hay fallos en el hardware del computador, tales como errores de paridad, falta de suministro de potencia, ... Son de muy alta prioridad. No son *enmascarables*, es decir, no se puede inhibir la ejecución de la rutina de interrupción correspondiente una vez se produce la interrupción.

Interrupción Software : producida por cualquier causa excepcional durante la ejecución de un programa: overflows, división por 0, código de operación ilegal, ...

Interrupción Temporal : son interrupciones cíclicas, generadas por el reloj del sistema. Tienen prioridad alta. Un uso común es la medición de tiempos en la CPU.

Interrupción de E/S : son generadas por los periféricos, para solicitar la atención de la CPU en ciertas situaciones. Por ejemplo, pueden indicar que ha terminado una acción de E/S (como localización de una posición en una cinta, posicionamiento del brazo móvil de un disco duro, etc, ...).

7.3.2. Operación

Supongamos que queremos leer un fichero del disco duro. A partir del nombre del fichero, el sistema operativo puede determinar en qué pista y sector comienza. Para empezar a leer bytes (su contenido) es necesario posicionar previamente el cabezal de lectura del HD en el lugar adecuado. El tiempo necesario para realizar esta operación (*tiempo de acceso*) es de varios milisegundos. Está claro que mantener la CPU sin hacer nada (o realizando una espera activa) durante varios milisegundos resulta en una pérdida de eficiencia del sistema.

Por tanto, resulta más interesante permitir que la CPU ejecute otras instrucciones mientras el controlador del HD se encarga de posicionar el cabezal. Cuando esta operación termine, el controlador enviará una señal de interrupción a la CPU. Ésta señal, activará algún bit de un registro interno a la CPU. Por su parte, la CPU chequea este bit antes de ejecutar una nueva instrucción y, si lo encuentra a uno, realiza la siguiente secuencia de operaciones:

1. Resetea el bit e indica al controlador que estamos atendiendo su interrupción.
2. Deshabilitar la posibilidad de que se nos pueda interrumpir de nuevo mientras estamos atendiendo a este periférico (no se vuelve a mirar el mencionado bit, aunque se ponga a uno).
3. Transferir el control a una posición **DirRTI** de memoria (donde se encuentra la rutina de tratamiento de la interrupción que estamos atendiendo), guardando previamente el contador de programa (**PC**) en **PCActual**.
4. Ejecutar la **rutina de tratamiento de interrupción**:
   - ❶ Salvar el estado del procesador (SR, puntero de pila, registros de propósito general, etc...)
   - ❷ Leer el fichero.
   - ❸ Restaurar el estado del procesador (recupera SR, puntero de pila, registros de propósito general, etc...).
5. Volver a habilitar las interrupciones.
6. Seguir la ejecución del programa por donde se quedó antes de ser inte-

rrumpido, es decir, recuperar el antiguo valor del $PC \leftarrow (PCActual)$.

7.4. ACCESO DIRECTO A MEMORIA

Aunque la técnica de interrupciones elimina el problema de la espera activa, aún queda para el procesador la tarea de ir leyendo datos de los dispositivos de E/S y de ir escribiéndolos en memoria. Ésta tarea es tan rutinaria y simple, que estamos desperdiciando la potencia de cálculo de un procesador si lo tenemos ocupado con este tipo de programas. La solución a este problema consiste en incluir en el sistema computador un dispositivo que se encargue de realizar estas transferencias: el dispositivo de **Acceso Directo a Memoria**(DMA).

7.4.1. Organización

En la figura 7.5 encontramos un diagrama de bloques con la configuración de un sistema que incluye DMA. Como puede observarse, el DMA, y al contrario que los demás dispositivos, puede leer y escribir tanto en el bus de datos como en el de direcciones. Este dispositivo va a funcionar como un esclavo de la CPU (o coprocesador para operaciones de E/S), realizando las tareas de transferencia de E/S que le indique la CPU. El significado de las señales lo veremos más adelante, al hablar de su modo de operación.

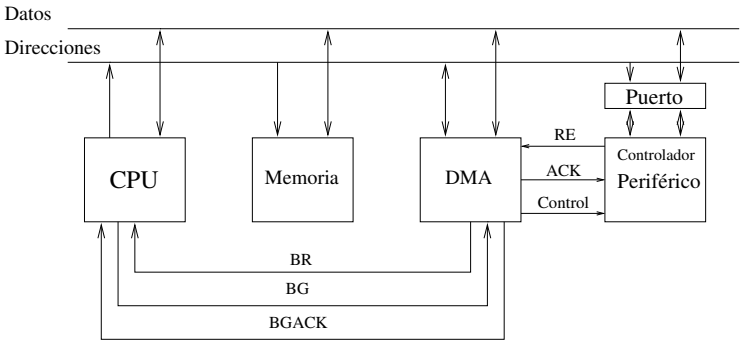


Figura 7.5: Esquema del DMA

7.4.2. Operación

Supongamos que queremos cargar un fichero en memoria. La CPU programará el DMA para que este dispositivo se encargue de la lectura del fichero y

su almacenamiento en memoria. Mientras, la CPU puede dedicarse a ejecutar otro tipo de programas. Cuando el DMA termina, lo indicará a la CPU (mediante una señal de interrupción) para que sepa que ya están disponibles en memoria los datos que pidió.

Con más detalle, son necesarios los siguientes pasos:

1. La CPU direcciona al DMA como si fuese un periférico más, para indicarle que quiere "hablar" con él. Por el bus de datos, la CPU le transfiere al DMA la información que necesita para realizar la transferencia, escribiendo para ello en los registros internos del DMA:
   - ❶ Dirección de memoria en la que almacenar el fichero.
   - ❷ Dirección, en el disco duro, en la cual se encuentra almacenado el fichero.
   - ❸ Sentido de la operación, es decir, si es lectura o escritura de HD (lectura en nuestro ejemplo).
   - ❹ Longitud del bloque a transmitir (número de bytes del fichero).
2. El DMA hace la transferencia, leyendo los datos del disco y direccionando posteriormente la memoria para escribir en ella cada byte. Para ello necesita el BUS (tanto de datos como de direcciones y, a veces, también el de control), que podrá usar en los momentos en que la CPU no acceda a la memoria (porque esté trabajando con registros, caché, coprocesador, etc.). Para pedir el bus a la CPU usa un protocolo de tipo *handshaking* ("estrechamiento de manos"), mediante la secuencia de señales: **BR** (*Bus Request*, petición del bus), **BG** (*Bus Grant*, cesión del bus) y **BGACK** (*Bus Grant ACKnowledge*, reconocimiento de cesión del bus). Tras esta última señal, el DMA dispone del bus para uso propio.
3. Cuando el DMA termine, mandará una interrupción a la CPU para indicarle que ha acabado y que ya puede disponer en memoria de los datos que le pidió.

A la hora de realizar las transferencias, existen varios modos de operar. Por ejemplo, dependiendo de cómo se repartan el uso de bus la CPU y el DMA tenemos:

- **Bloques sin ruptura**: si el DMA tiene el bus, no lo suelta hasta que termina la transferencia de un bloque completo. La CPU tiene que esperar a que el DMA libere el bus.
- **Ciclo a ciclo**: DMA y CPU se reparten ciclo a ciclo el control de bus (uno para la CPU y otro para el DMA, alternando).
- **Ciclos muertos de CPU**: el DMA solo tomará control del bus cuando se dé cuenta de que la CPU no lo está utilizando.

- **Controlado por el periférico**: el periférico es el que decide cuando se hace la transferencia.

Otro dispositivo más potente para realizar operaciones de E/S son los **Canales de Entrada/Salida**, que son coprocesadores, con varias DMAs en su interior, que pueden ser programados, y con más inteligencia que los DMAs.

SINOPSIS

Hemos visto la evolución de la Entrada/Salida desde la E/S programada hasta el DMA, mencionando también los Canales de E/S. Hemos podido comprobar como un incremento del hardware, con su consecuente aumento del coste del sistema computador completo, permite realizar operaciones de comunicación de E/S cada vez más eficientes.

8 Introducción a los Sistemas Operativos

OBJETIVOS

- Identificar las tareas de las que es responsable el Sistema Operativo (S.O.) en un computador
- Comprender los mecanismos implicados en la gestión de sistemas de tiempo compartido, multiusuario y multitarea
- Completar con el nivel de memoria virtual la descripción de sistemas con jerarquía de memoria

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

8.1. INTRODUCCIÓN

Un **Sistema Operativo** (S.O.) se puede definir como un conjunto de programas que permiten utilizar el hardware de la máquina, a uno o varios usuarios, de una manera sencilla y eficiente.

Vamos a ver un poco más detenidamente esta definición. Un S.O. es un *conjunto de programas*, es decir, algo *Software* o *Firmware* (software que no se puede modificar, como microsubrutinas o programas residentes en ROM), cuya misión es facilitar la gestión del *hardware*. En principio, se dedicará al control de los dispositivos de Entrada/Salida, como pueden ser el disco duro, CD-ROM, vídeo,... El procesador también es parte del hardware, por lo que será otro recurso más a gestionar, sobre todo cuando hay más de un usuario usándolo a la vez.

El S.O. controla todo el hardware con el objetivo de que la máquina dé un rendimiento óptimo, de modo que, si un usuario o aplicación quiere utilizar el hardware, será mediante llamadas a las rutinas del S.O. Se habla de una estructura en capas (figura 8.1), en la que el S.O. oculta los dispositivos hardware (disco, puertos de E/S...), proporcionando a las aplicaciones (procesos) un interfaz más sencillo. El proceso solicita estos servicios haciendo llamadas al S.O., que maneja el hardware mediante el uso de *interrupciones*.

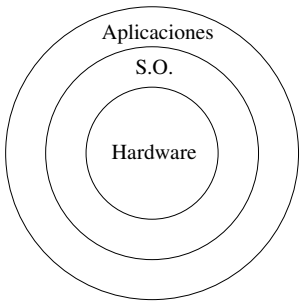


Figura 8.1: Capas de un S.O.

8.1.1. Tareas que realiza un Sistema Operativo

Un S.O. debe suministrar los medios para poder llevar a cabo las siguientes operaciones:

- **Concurrencia** o capacidad de poder realizar varias actividades simultáneas, como el solapamiento de actividades de E/S con computación en

el procesador.
- **Interfaz con el usuario**, es decir, establecer los mecanismos de comunicación entre el usuario y el hardware.
- **Compartición de recursos** ya que las actividades concurrentes, que se ejecutan a la vez, pueden compartir recursos hardware e información. Entre estos recursos se incluye procesador(es), memoria, E/S y comunicaciones (p.e. conexión a la red).
- **Almacenamiento** a largo plazo (sistema de ficheros) dada la necesidad de los usuarios de guardar sus datos en el computador para usos posteriores.
- **Protección**, pues, aunque habrá elementos que compartir, otros serán privados y deberán ser de acceso restringido.
- El sistema operativo ha de responder de forma adecuada, independientemente de las situaciones impredecibles que pudieran tener lugar. Es importante que el S.O. pueda realizar **recuperación de errores**.

Además de implementar estas funcionalidades, sería deseable que el S.O. fuera *eficiente, fiable, fácil de corregir o modificar*, y que *no ocupe excesivos recursos* (p.e. espacio en disco duro, memoria, etc.).

8.1.2. Evolución de los Sistemas Operativos

La historia de los Sistemas Operativos se puede dividir en una serie de generaciones, cada una caracterizada por introducir alguna novedad.

Las primeras máquinas carecían de S.O. Los usuarios programaban en código máquina y todas las funciones tenían que ser programadas por ellos mismo a mano (1940-50). Además, se perdía mucho tiempo entre la conclusión de un trabajo y el comienzo del siguiente. De forma sintética los pasos en la evolución de los S.O han sido:

1. Creación de los primeros S.O. (1950) constituidos por programas a los que se les pasaban un conjunto de trabajos por *lotes* o *batch* (.BAT en el *MS-DOS*). El S.O. estaba diseñado para ir ejecutando cada uno de los trabajos uno a uno. Al terminar cada trabajo (normal o anormalmente), el control se devolvía automáticamente al S.O., que iniciaba el trabajo siguiente.
2. Con la introducción de la *multiprogramación* en los años 60, la entrada/salida se comenzó a realizar en paralelo con la computación, aprovechándose mejor el procesador.
3. Aparición del *procesamiento en tiempo compartido*, donde la máquina es

capaz de ejecutar programas de varios usuarios. Ésto se hace de modo que no sólo se comparten los recursos, sino que además se tiene la sensación de que la máquina la estamos usando sólo nosotros. Así se explota el procesador de un modo en el que se obtiene un mayor rendimiento. Para ello, el S.O. asigna pequeños intervalos de tiempo de CPU ("time slices" o *quantum* de tiempo) a cada usuario.

4. La complejidad de los S.O. crece día a día, especialmente con los sistemas *multiprocesador*, en los que se debe dividir eficientemente las tareas a realizar entre los procesadores que tenga disponibles el computador.

8.2. ADMINISTRACIÓN Y PLANIFICACIÓN DE PROCESOS

8.2.1. Proceso

Proceso es cualquier programa que el S.O. haya cargado e iniciado, asignándole los recursos del sistema que necesite (espacio de direcciones de memoria virtuales y físicas, variables de entorno, archivos, ...), y aspire a ser ejecutado en la CPU cuando sea posible. Observar que el S.O. se compone de varios programas que se ejecutan y son, por tanto, procesos.

El sistema operativo necesita almacenar ciertos datos sobre cada proceso, para poder realizar sobre ellos ciertas operaciones que veremos más adelante. La estructura de datos que se usa para guardar la información relativa a un proceso se denomina **PCB** (*Bloque de Control de Proceso*) y contiene, entre otros, los siguientes valores:

- Estado actual del proceso
- Identificador (*PID*, un número) que es único para cada proceso
- Posición de memoria en donde se encuentra el código del proceso
- Identificación del proceso *padre* (el que lo creó)
- Identificación de los procesos *hijos* (los que él cree)
- Información de los recursos que le han sido asignados
- Prioridad del proceso, etc...

La forma en que el sistema operativo gestiona los programas en ejecución dependerá de si éste permite o no tener varios procesos usando el procesador simultáneamente (*procesamiento en tiempo compartido* o *procesos concurrentes*):

- Un S.O. *monotarea* y *monousuario*, como MS-DOS, sólo carga en memoria un único programa, pasándole el control a éste, que se ejecutará de

forma exclusiva en el procesador. El programa cargado será el responsable devolver el control al S.O cuando termine su ejecución. En MS-DOS, al conjunto de interrupciones encargadas de ofrecer al programa de usuario los servicios de E/S se le denomina BIOS (*Basic Input Output System* o sistema de E/S básico).

- Un S.O. *multitarea* es capaz de ejecutar a la vez varios procesos. Para ello, hemos de disponer de una interrupción temporal que se active periódicamente. Con esta interrupción, se invocará al S.O., que conmutará el uso de la CPU entre los diferentes procesos que se estén ejecutando. De esta manera, se da la impresión de que se están corriendo simultáneamente varios programas en la CPU. El *quantum* de tiempo o *time-slice* es el tiempo que transcurre desde que el procesador empieza a atender a un proceso hasta que deja de hacerlo, para pasar a tratar el siguiente. Cada uno de los procesos se comporta como si tuviera su propio puntero contador de programa (PC) y puntero de pila (SP).

UNIX y VMS son ejemplos de sistemas operativos que hace ya mucho tiempo que ofrecen multitarea.

Sistemas operativos como Windows 3.x ofrecen lo que se denomina multitarea *cooperativa* o *no-prioritaria* que no es una verdadera multitarea. En este esquema es cada tarea la que es responsable de pasar el control al núcleo del sistema operativo para que conmute a otra. Si una tarea no hace esto es imposible conmutar (pudiendo quedar colgado).

En Windows 9x, NT la multitarea es *prioritaria* es decir se realiza una conmutación entre procesos cada cierto tiempo. Además existe un proceso encargado de emular la ejecución de todas las aplicaciones de Windows 3.x que implementan la multitarea *no-prioritaria*. Este proceso sólo se ejecuta durante su *time slice*.

Un término que encontramos en los S.O. actuales es el de *thread*, que encontramos traducido por *subproceso*, *hebra* o *hilo*. Con un significado análogo al de proceso, los *threads* tienen la propiedad de poder compartir recursos entre sí, como por ejemplo el rango de direcciones de memoria asignadas.

8.2.2. Estados de un Proceso

En un sistema multitarea, un proceso se puede encontrar en tres estados distintos:

a) *RUNNING* (en ejecución): cuando el proceso se está ejecutando (está siendo atendido por la CPU, o usando la CPU).

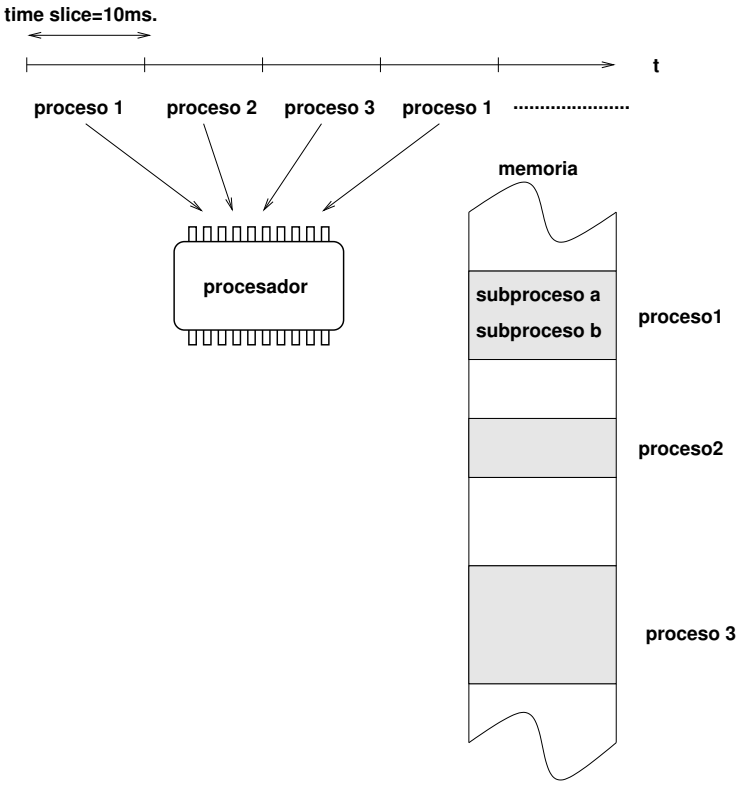


Figura 8.2: S.O. Multitarea

b) *READY* (listo): el proceso podría estar en ejecución si la CPU estuviese libre, es decir, si ésta no estuviese ejecutando otro proceso.

c) *BLOCKED* (bloqueado): el proceso está a la espera de que ocurra algún evento, normalmente alguna transacción de Entrada/Salida.

En sistemas *monoprocesador* sólo puede haber un proceso *running*, pero varios *ready* y *blocked*. El sistema operativo tiene una lista con los procesos *ready* (ordenados por prioridad) y otra con los procesos *blocked* (en este caso la lista está desordenada).

Las transiciones entre estos estados se producen mediante llamadas del sistema operativo a una serie de funciones, con el nombre del proceso como parámetro, excepto en el caso del paso de *running* a *blocked*, que es provocada por el propio proceso cuando necesita realizar una operación de E/S. Las posibles transiciones entre estados de un proceso aparecen representadas en la figura 8.3 y se comentan a continuación.

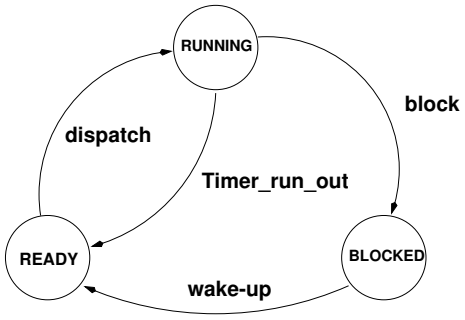


Figura 8.3: Estados de un proceso

- *Dispatch*: se produce cuando un proceso toma posesión de la CPU, debido a que el que la estaba usando ha agotado su tiempo de uso (*quantum*) o ha terminado su ejecución. El proceso que se apropiará de la CPU será el primero de la lista de procesos *ready*. El sistema operativo deberá almacenar el nuevo estado de los procesos del sistema, pues se ha producido lo que se denomina un *cambio de contexto* (cambio del proceso en ejecución).
- *Timer_run_out*: el sistema operativo asigna a cada proceso un tiempo fijo (*quantum*) de uso de la CPU. Si en ese tiempo no es capaz de terminar lo que le quedaba cuando tomó posesión de la CPU la última vez, se activa el *Timer_run_out* (una interrupción temporal), con la que se obliga al proceso que estaba *running* a abandonar la CPU, pasando a estado *ready*.
- *Block*: es el otro motivo por el que un proceso puede dejar libre la CPU. Cuando un proceso llega a una instrucción que ha de esperar a que se complete una operación de E/S, éste (para evitar estar ocupando la CPU durante la operación de E/S) pasa a estado *blocked* y libera la CPU. Esta transición la provoca el propio proceso cuando debe esperar a que se produzca algún evento de E/S.
- *Wake_up*: esta señal se envía a un proceso que está *blocked* cuando se ha producido el evento de E/S que le hizo pasar a este estado. Como ya no tiene que esperar nada más para continuar su ejecución, pasa a la lista de procesos *ready*, esto es, listos para seguir su trabajo.

Vamos a ver un par de ejemplos de cómo funciona esto. Supongamos que tenemos tres procesos, que llamaremos A, B y C, que se van a ejecutar concurrentemente usando la misma CPU. El quantum (Q) de tiempo que asigna el

sistema operativo a cada proceso será de 10 milisegundos. Veremos que ocurre a lo largo del tiempo para las dos situaciones que aparecen en la figura 8.4.

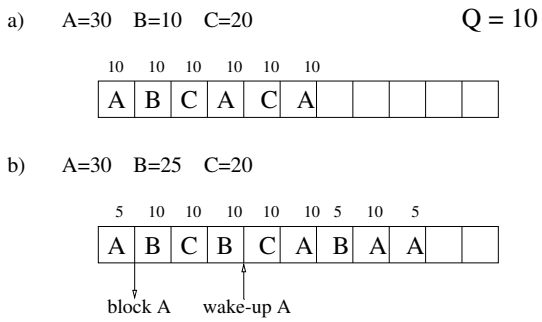


Figura 8.4: Ejemplos de transición entre estados de un proceso

En el primer ejemplo, los procesos tardan 30, 10 y 20 milisegundos respectivamente en ejecutarse. En este caso no habrá operaciones de Entrada/Salida, por lo que cada proceso irá tomando durante 10 mseg. la CPU, realizará las operaciones que pueda en ese tiempo y dejará libre el procesador para el siguiente proceso. Los diferentes procesos, al no haber ninguno prioritario, irán apropiándose de la CPU cíclicamente, mientras no hayan terminado su trabajo (ver figura 8.4.a).

Para el segundo caso, vamos a suponer que los procesos van a necesitar utilizar 30, 25 y 20 milisegundos la CPU. Ahora, cuando el proceso A lleve 5 milisegundos, va a tener que leer de un fichero y no podrá seguir su ejecución hasta tenerlo cargado en memoria, por lo que el proceso pasará a estado *blocked*. Previamente, antes de bloquearse, programará el DMA para que realice la transferencia y, cuando ésta termine (30 milisegundos después), el DMA avisará mediante una interrupción. Cuando esto ocurra, el proceso pasará a la lista de procesos *ready*, para que continúe su ejecución en el momento que le corresponda usar la CPU. La figura 8.4.b representa esto esquemáticamente.

Al hablar de los posibles estados de un proceso y sus transiciones, hemos visto como cuando se produce el *Timer_run_out*, realmente lo que está ocurriendo es una *interrupción temporal*. Desde el punto de vista de un S.O., la *interrupción* es un suceso que altera la secuencia normal de ejecución de un proceso. Cuando se activa una interrupción de éste tipo, se llevan a cabo una serie de acciones:

❶ El sistema operativo toma el control
❷ El S.O. salva el estado del proceso en ejecución (en el PCB)

❸ El S.O. analiza la interrupción y le pasa el control al proceso (rutina) adecuado de manejo de esa interrupción
❹ El proceso de tratamiento de la interrupción realiza su trabajo
❺ El S.O. restaura el estado del proceso interrumpido y éste continúa ejecutando por donde se quedó

Las interrupciones se pueden anidar, es decir, un proceso de tratamiento de interrupción puede a su vez ser interrumpido y así sucesivamente.

Cada vez que un proceso tiene que salir de la CPU para que entre otro, se produce lo que se denomina un *cambio o conmutación de contexto*, que consiste en salvar el estado del proceso en ejecución, antes de que entre el nuevo proceso.

8.2.3. Planificación

La *planificación* (*processor scheduling*) es el mecanismo, que implementan los S.O., para asignar procesos a los procesadores. El *planificador* (scheduler) da las *prioridades* a los procesos en la cola de procesos *ready*. Hay dos formas en las que se puede asignar un procesador a un proceso:

No Preemptive (no expropiativa, no prioritaria o cooperativa) → si un proceso toma la CPU, nadie le podrá interrumpir hasta que termine, o le ceda el control al S.O.

Preemptive (expropiativa o prioritaria) → en *tiempo compartido*, cuando transcurre el *quantum* de tiempo, se producirá una interrupción del reloj, saliendo de la CPU el proceso que la tenía y entrando en su lugar el primero en la lista de procesos *ready*.

La forma en que el *scheduler* asigna las prioridades a los diferentes procesos de la lista de procesos *ready* da lugar a varios **algoritmos de planificación**, que pasamos a ver a continuación:

- **FIFO (cola)**, el primero que llega es el primero en salir, es de tipo *no preemptive*.
- **Round Robin (RR)**, es una FIFO *preemptive*.
- **Shortest-Job-First (SJF)** da más prioridad a los procesos más cortos. Con esta política se consigue una mayor razón de ejecución de trabajos.
- **Highest-Response-Ratio-Next (HRN)** (el siguiente con *relación de respuesta* máxima). Las prioridades que se asignan son dinámicas, con lo que mejoramos el algoritmo *SJF*. La prioridad en un momento determinado viene dada por la expresión:

$$prioridad = \frac{tiempo\_esperando + tiempo\_de\_ejecución}{tiempo\_de\_ejecución}$$

En principio, los procesos cortos tienen mayor prioridad. Los procesos largos, al tener que esperar más, irán aumentando poco a poco su prioridad.

8.2.4. Operaciones sobre Procesos

Sobre un proceso se pueden realizar un gran número de operaciones, como crear, destruir, suspender, reanudar, cambiar la prioridad, bloquear, despertar, etc...

Crear. Cuando se crea se realizan una serie de operaciones:

- Dar un nombre al proceso
- Introducirlo en la lista de procesos conocidos
- Determinar la prioridad inicial
- Crear su PCB
- Asignar los recursos iniciales

Un proceso puede a su vez crear otro(s) proceso(s). Al proceso creador se le llama proceso *padre* y a los creados procesos *hijo(s)*.

Aniquilar. Cuando esto sucede, se borra del sistema, se liberan los recursos que hubiese ocupado y se elimina su PCB correspondiente.

Suspender y Reanudar. Cuando se observa alguno de los síntomas siguientes en un sistema, se suele *suspender* el proceso(s) que se sospecha que es el causante:

- El sistema va lento
- Un proceso actúa de forma sospechosa
- El sistema está muy cargado

Cuando se vuelve a la normalidad, se suele *reanudar* a los procesos que estuviesen suspendidos.

Núcleo del S.O. (KERNEL)

Hay una parte del código del sistema operativo que es la que más se suele usar. Esta parte reside en memoria principal y se denomina núcleo del sistema operativo.

Entre otras, las funciones más importantes de este código son:

- Crear y destruir procesos

- Dispatching, suspensión, reanudación,... de procesos
- Manejar los PCB de los procesos (y gestionar los cambios de estado en general)
- Manejar interrupciones y la sincronización
- Realizar actividades de Entrada/Salida
- Asignar y liberar recursos (p.e. memoria)
- Soportar el sistema de ficheros

8.3. MEMORIA VIRTUAL

Hemos visto como el S.O. es el encargado de gestionar los recursos del sistema. Entre ellos, la memoria es uno de los más importantes que debe administrar. Imagínese que quisiésemos usar un rango de direcciones de memoria mayor del que físicamente tenemos, sin incrementar el hardware. Las posibles soluciones propuestas por orden cronológico, en la historia de la computación, son:

- *Conmutar bancos de memoria* entre sí, controlándolos con un registro, de forma que en un momento dado las direcciones sólo se refieren al banco seleccionado. Es una solución lenta y cara.
- *Overlays (recubrimientos).* Son definidos por el programador. En ellos, la información se va cargando conforme se va necesitando, no de una sola vez al principio del programa (en *MS-DOS* son los ficheros con extensión .OVL).
- *Memoria Virtual.* En este caso se usa la memoria principal como "caché" de la memoria secundaria (disco duro).

La *memoria virtual* surge, por tanto, por la necesidad de ejecutar programas que usan una cantidad de memoria superior a la que se tiene físicamente. Para ello, habrá que desarrollar mecanismos eficientes para la compartición de la memoria principal.

Al trabajar con la memoria virtual, nos vamos a encontrar con una terminología básica:

- *Página*: bloques elementales con los que se trabaja en las transferencias entre memoria principal y secundaria.
- *Fallo de Página (miss o page fault)*: se produce cuando el programa hace referencia a una dirección de memoria que no se encuentra entre las direcciones actualmente cargadas en memoria principal. En este caso, habrá que traer de memoria secundaria la página que la contenga. Si en memoria principal no quedase sitio suficiente para almacenarla, se deberá elegir (siguiendo alguna política) la página a la que reemplazará.

Si esta hubiese sido modificada por el programa, se deberá salvar la copia actualizada en el disco.

- *Dirección Virtual / Dirección Física*. La dirección virtual es la que usan los programas, dentro del espacio de direcciones de la memoria virtual. La dirección física es la dirección en la que realmente se almacenan los datos en memoria principal.
- *Traducción de Direcciones (memory mapping)*: es el mecanismo por el que asignamos a una dirección virtual una dirección física.

A la hora de diseñar una memoria virtual, los aspectos a tener en cuenta vienen motivados por el alto coste de un fallo de página. Los más importantes son:

- Tamaño de la página (mismas consideraciones que el tamaño de línea en la caché), suele ser de 4K ó 16K.
- Gestión de los fallos de página. Son tratados por software, pues hay tiempo suficiente para ello y de este modo se puede hacer de un modo más inteligente.

8.3.1. Ejemplo

Vamos a ver, para terminar, un ejemplo con valores numéricos de un sistema de memoria virtual. El sistema de nuestro ejemplo va a trabajar con páginas de 4 Kbytes de tamaño. La memoria física tendrá un tamaño de 4 Megabytes y el espacio de direcciones virtual será de 4 Gigabytes. En este caso:

$$\text{Tamaño de página} = 2^{12} = 4Kbytes = P$$
$$\text{Tamaño de Mem. Real} = 2^{22} = 4Mbytes = M \rightarrow 2^{10} \text{ páginas}$$
$$\text{Tamaño de Mem. Virtual} = 2^{32} = 4Gbytes = V \rightarrow 2^{20} \text{ páginas}$$

En la figura 8.5 podemos ver los campos en que se dividen las direcciones, tanto virtuales como físicas. En ambos casos, los 12 bits menos significativos se usan para dar el desplazamiento dentro de la página. El espacio de direcciones virtuales tiene 2^{20} páginas, mientras que el físico sólo consta de 2^{10}, por lo que se deberá establecer un mecanismo para traducir las páginas virtuales en físicas. Para ello se usa una tabla como la que puede observarse en la figura 8.5.

Esta tabla tendrá 2^{20} entradas, una por cada página de memoria virtual, y cada entrada nos dirá si la página en cuestión se encuentra cargada en memoria y, en este caso, en qué posición física (para indicarlo necesitará 10 bits), o si, por el contrario, está almacenada en memoria secundaria y dónde. En caso de que se encuentre en memoria principal, un bit de modificación M

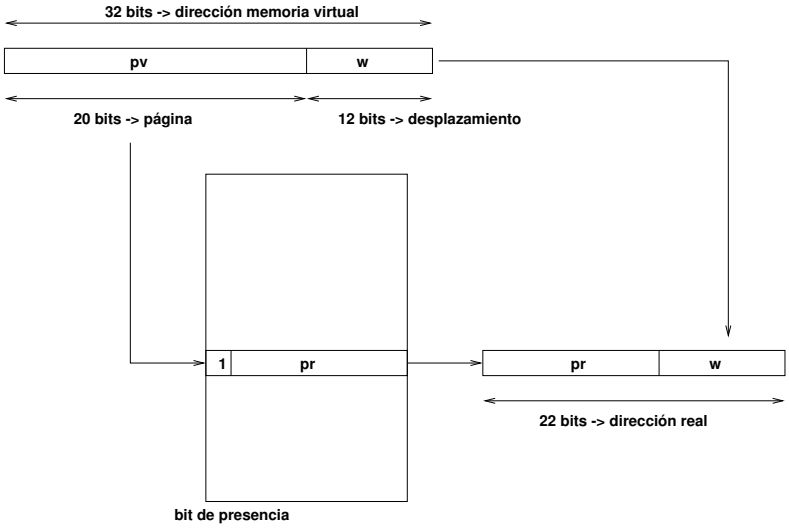


Figura 8.5: Traducción de referencia virtual a real

nos indicará si la página en memoria ha sido modificada o se encuentra como en el momento en que se trajo a memoria principal.

Como se puede apreciar en este ejemplo, utilizar una tabla con tantas entradas como páginas virtuales da lugar a grandes tamaños para la tabla. Para paliar este problema, se riza un poco más el rizo y se pagina a su vez la tabla de páginas. Es decir, la tabla se parte en páginas y sólo tendremos en memoria las páginas de la tabla que estén siendo utilizadas. Siguiendo con el mismo ejemplo, para un espacio de direcciones virtual de 32 bits, una posible implementación para la tabla de traducción aparece representada en la figura 8.6.

Podemos observar como el cálculo de la dirección física se hace en dos fases. En un primer paso usamos los 8 bits más significativos para elegir la tabla de traducción a usar dentro de un *directorio de tablas*. A partir del valor así obtenido, que será la dirección base de la tabla de traducción a usar, los siguientes 12 bits nos seleccionarán la entrada adecuada dentro de la tabla de traducción. Esta tabla nos dará como salida la dirección base de la página que estamos buscando, por lo que si le sumamos los 12 bits restantes (desplazamiento dentro de la página), obtendremos la posición física correspondiente a nuestra dirección virtual.

Este método presenta una serie de problemas:

a) Son necesarios dos accesos a memoria para realizar la traducción, por lo

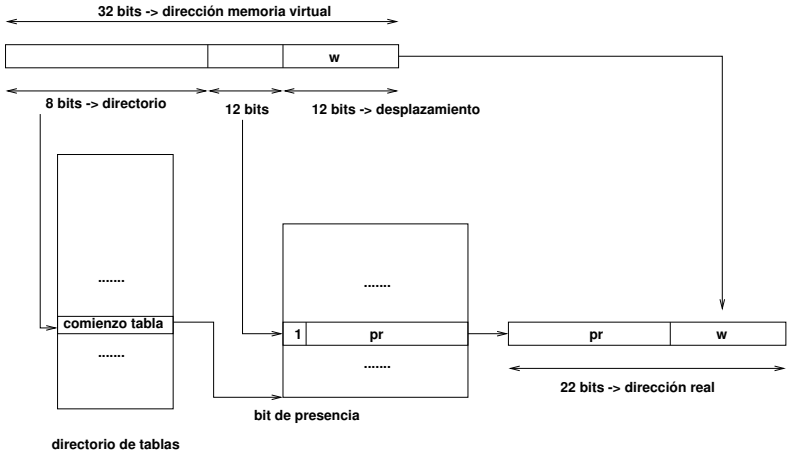


Figura 8.6: Traducción paginada

que será más lenta. Se puede usar una **TLB (Translation Lookaside Buffer)** , a modo de memoria caché, que contenga los pares (*dirección virtual, dirección física*) más utilizados, acelerando así el proceso.

b) El propio acceso a la tabla, dado que está paginada, puede producir un fallo de página, con la penalización que eso lleva consigo.

SINOPSIS

Gran parte del rendimiento ofrecido por un computador recae en el buen diseño del sistema operativo que utilice. El sistema operativo, aunque software, es fundamental en el uso de los elementos hardware que componen la máquina. Sólo hay que fijarse un poco para observar que es un producto clave en el mercado informático, tanto más cuando de él dependen recursos como la conectividad en red que cada día cobra más importancia. En este capítulo hemos esbozado las características generales de un sistema operativo, incidiendo en dos aspectos básicos: la gestión de procesos y la memoria virtual.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Apéndice A

Sistemas de representación especiales

A.1. CÓDIGOS DETECTORES/CORRECTORES DE ERRORES

La aparición de errores en el almacenamiento o transmisión de la información ha dado lugar a la creación de códigos que permiten corregir o al menos detectar los fallos que ocurren en el proceso de almacenamiento o transmisión. Dada una secuencia binaria `I` que contiene la información, se procede a añadir una secuencia redundante `R` deducida a partir de `I` que nos permitirá la detección y/o corrección de los errores que se produzcan en `I`.

Se define la distancia Hamming entre dos palabras binarias, `d(x,y)`, como el número de bits en que difieren. Se entiende por distancia Hamming de un código de palabras binarias, `d(C)`, al $min\{d(x,y)/x,y \in C, x \neq y\}$. El campo redundante `R` se añade de forma que el código resultante formado por las nuevas palabras `(I,R)` tenga una distancia mayor que el código formado por las palabras `I` y por tanto permita distinguir si ha habido errores.

Se demuestra que el código con información redundante ha de tener una distancia mínima de $k+1$ si se quieren detectar k errores de un bit; y que dicha distancia ha de ser $2k+1$ si lo que se quiere es poder corregir k errores de un bit.

A.1.1. Bit de paridad

Es un mecanismo simple para la detección de un bit erróneo. Consiste en añadir 1 bit de redundancia calculado de la siguiente forma:

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

- Paridad par: El número total de 1's en la palabra binaria resultante `(I,R)` ha de ser par.
- Paridad impar: El número total de 1's en la palabra binaria resultante `(I,R)` ha de ser impar.

A.1.2. Códigos polinómicos

El campo redundante `R` se calcula a partir del campo de información `I` y una palabra de control `C`. Los bits redundantes se determinan como $R = I mod C$ expresando R con la misma longitud en bits que C. La facilidad del cálculo de esta operación es un elemento a favor de estos códigos redundantes.

Estos códigos, tambien denominados de redundancia cíclica (CRC), se suelen utilizar en transmisiones en las que errores en ráfagas de bits son frecuentes.

Como ejemplo, si $I = 1010001100$ y $C = 1010$ tendremos $R = I mod C = 0010$. Se suele expresar C como un polinomio con los coeficientes igual a los dígitos binarios de C, en este ejemplo $C \equiv x^3 + x$.

A.1.3. Código Hamming

Con este código se pretende poder corregir y detectar errores de un bit. Se parte de un campo de información `I` de n bits al que añadiremos p bits redundantes para producir una palabra de $n + p$ bits que es la que será transmitida. El número de bits redundantes se determina como el menor entero sujeto a la condición $2^p \geq n + p + 1$. Veamos este código para el caso de $n = 4$.

Para $n = 4$, tenemos que el número de bits a añadir es $p = 3$. El campo de información lo denotamos $I = i_4 i_3 i_2 i_1$, y el campo redundante $R = p_3 p_2 p_1$. La palabra de $n + p$ bits resultante entremezcla los bits de `I` y `R` de la siguiente manera: $B = b_7 b_6 b_5 b_4 b_3 b_2 b_1 = i_4 i_3 i_2 p_3 i_1 p_2 p_1$. Aquí B es la palabra a transmitir o enviar. Los bits redundantes se colocan en las posiciones potencia de dos de B ($b_{2^{k-1}} = p_k$).

Codificación: El bit redundante p_i es calculado como la operación OR exclusiva de los bits b_k ($1 \leq k \leq 7$), que pertenezcan a `I`, cuyo subíndice k expresado en binario contenga el bit i igual a 1. De esta forma:

$$\begin{aligned} p_1 &= b_3 \oplus b_5 \oplus b_7 \\ p_2 &= b_3 \oplus b_6 \oplus b_7 \\ p_3 &= b_5 \oplus b_6 \oplus b_7 \end{aligned}$$

Decodificación: Recibida la palabra $B = b_7b_6b_5b_4b_3b_2b_1$ se determina la palabra $r_3r_2r_1$ donde cada r_i se calcula como la operación OR exclusiva de los bits b_k $(1 \leq k \leq 7)$ de B, cuyo subíndice k expresado en binario contenga el bit i igual a 1; es decir se calculan como los p_i pero teniendo en cuenta todos los bits de B incluyendo los redundantes, y no sólo los de información. Así:

$$\begin{aligned} r_1 &= b_1 \oplus b_3 \oplus b_5 \oplus b_7 \\ r_2 &= b_2 \oplus b_3 \oplus b_6 \oplus b_7 \\ r_3 &= b_4 \oplus b_5 \oplus b_6 \oplus b_7 \end{aligned}$$

La clave del código radica en los siguiente: Tal como hemos construido los bits redundantes si existe un error en el bit b_k se activarán aquellos r_i que se correspondan con los lugares donde k expresado en binario tiene 1's. Es decir, la palabra $r_3r_2r_1$ considerada un número binario positivo indica el lugar en B donde se ha producido el error, con lo cual no solo está detectado sino que cómo sabemos qué bit es podemos corregirlo.

A.2. CÓDIGOS DE LONGITUD VARIABLE

Al construir un mensaje con símbolos de un alfabeto podemos observar que no todos los símbolos del alfabeto aparecen con igual frecuencia (probabilidad). De ahí surge la idea que será más eficiente una codificación que emplee menos bits para representar los símbolos más frecuentes. Esta es la idea que existe detras de los códigos usados en la compresión de mensajes (por ejemplo los compresores de ficheros).

A.2.1. Código Huffman

Un código representativo de este tipo de codificaciones de longitud variable es el código Huffman.

La generación de este código sigue el siguiente esquema: Se parte de las frecuencias relativas (probabilidades) de cada símbolo del alfabeto. Los símbolos se ordenan por probabilidades decrecientes. De forma sucesiva se colapsan los dos símbolos con probabilidades más bajas en un núevo símbolo cuya probabilidad es la suma de los símbolos de los que proviene. El nuevo conjunto resultante se vuelve a reordenar y se repite el proceso hasta que sólo queda un símbolo con probabilidad uno. Los sucesivos pasos se traducen en un árbol binario que es usado para generar los códigos.

EJEMPLO: Sea el alfabeto {1,2,3,4,5} con probabilidades asociadas {0.4, 0.2, 0.25, 0.1, 0.05}.

```
Reordenación: 1(0.4) 3(0.25) 2(0.2) [4(0.1) 5(0.05)]
Agrupamiento: 1(0.4) 3(0.25) [2(0.2) A(0.15)]
Agrupamiento: 1(0.4) 3(0.25) B(0.35)
Reordenación: 1(0.4) [B(0.35) 3(0.25)]
Agrupamiento: 1(0.4) C(0.6)
Reordenación: [C(0.6) 1(0.4)]
Agrupamiento: D(1)
```

El árbol resultante se muestra en la figura A.1. Hasta llegar a los nodos símbolos desde el raíz, se va asignando un 1 a la rama derecha y un 0 a la izquierda.

Con lo cual el código resultante es:

SÍMBOLO	CÓDIGO
1	0
3	10
2	110
4	1110
5	1111

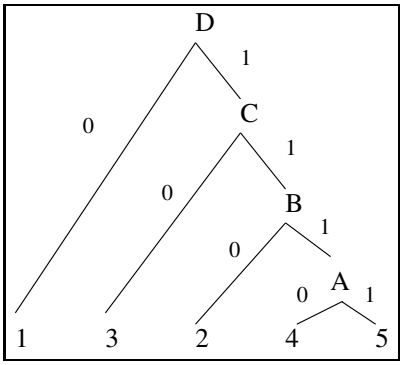
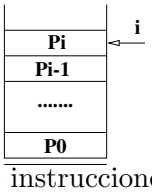


Figura A.1: Arbol de generación Huffman

A.3. EJERCICIOS

1. Elaborar los codigos Hamming para cada uno de los enteros naturales en binario menores a 31.
2. Utilice el código Huffman para comprimir el siguiente mensaje en octal: `7356735671357225225105252525252525252525525525555222377776154 35`. Calcule la longitud media resultante y compárela con la que se necesitaría si se usara binario natural.
3. Construya el diagrama de estados para un autómata de estados finitos que sea capaz de reconocer el código Huffman creado para el ejercicio anterior.
4. Usando el polinomio $C = x^{15} + x^{12} + x^5 + 1$ calcule el campo CRC para la secuencia binaria obtenida en 2.
5. Un procesador con arquitectura de 'cero direcciones (operandos)' ejecuta el programa:

```
1  push A      6  push B      11 pop R
2  push B      7  push C
3  inc         8  push A
4  add         9  add
5  pop A       10 mult
```



Resumimos a continuación la acción de las instrucciones, número de bytes en memoria que ocupan y duración del ciclo de instrucción (ck). Denotaremos P_i el dato que se encuentra en la cima de la pila, siendo i el puntero de pila. Cada vez que introducimos un nuevo dato en la pila i se incrementa en 1.

Instrucción	Acción	N.º bytes	ck
push X	$i \leftarrow i+1$ y después $P_i \leftarrow X$	3	75 ns.
pop X	$X \leftarrow P_i$ y después $i \leftarrow i-1$	3	75 ns.
add	$P_{i-1} \leftarrow P_i + P_{i-1}$ y después $i \leftarrow i-1$	1	75 ns.
mult	$P_{i-1} \leftarrow P_i * P_{i-1}$ y después $i \leftarrow i-1$	1	100 ns.
inc	$P_i \leftarrow P_i + 1$, i no varía	1	75 ns.

a) Calcular el valor almacenado en R, en función de los valores iniciales de las variables $A = A_0$, $B = B_0$ y $C = C_0$, cuando finaliza la ejecución del programa.

b) Sabiendo que todas las instrucciones del procesador tienen el mismo tamaño de código de operación, ¿cuál es el tamaño de los operandos?

c) Se desea diseñar un formato de código de operación variable basado en la codificación Huffman, en el que las instrucciones que aparecen con mayor frecuencia en un programa tengan menos bits en el código de operación, permitiendo reducir el tamaño medio de los programas. Realizar el diseño completo de dicho código de operación para el conjunto de instrucciones que se ha presentado sabiendo que el código que se analiza representa bastante bien el comportamiento general de los códigos escritos para este procesador.
Escribe los códigos de operación resultantes en la siguiente tabla:

Instrucción:	push	pop	add	mult	inc
C.OP. Huffman:					

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Apéndice B

Rendimiento de un computador

B.1. LEY DE AMDAHL

El término *cuello de botella* - del inglés *bottleneck* - se utiliza en la jerga del rendimiento de computadores para referirse al subsistema o subsistemas que degradan el rendimiento del equipo en general.

Puesto que todos los componentes de un computador están interconectados, un cambio en las prestaciones de un subsistema tiene un impacto inmediato en el rendimiento del sistema en general. Ahora bien, puestos a mejorar algo, el sistema se beneficiará más si mejoramos el subsistema donde está localizado el cuello de botella y, de forma inversa, si queremos abaratar costes, lo mejor es penalizar el subsistema con mejor rendimiento respecto a los demás.

Por otra parte, como no todos los componentes de un ordenador tienen la misma frecuencia de uso en la ejecución de un programa, resultará más beneficioso mejorar un subsistema cuanto más utilizado sea éste.

Como métrica del rendimiento en tiempos de un subsistema se emplea la denominada aceleración o ganancia en tiempos, conocida por su término anglosajón *speed-up*, definida como cociente el cociente de tiempos empleado por el subsistema a evaluar y una referencia:

$$\alpha = \frac{\text{Tiempo consumido}}{\text{Tiempo de referencia}}$$

Para cuantificar el efecto en rendimiento global de un sistema a partir del rendimiento de sus partes se utiliza la expresión conocida como **ley de Am-**

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

dahl, que establece que *la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.* De forma analítica, esta ley se resume en la siguiente fórmula:

$$\alpha = \frac{1}{(1-p) + \frac{p}{\alpha_m}}$$

cuyos parámetros poseen el siguiente significado:

- α: Es la aceleración global del sistema completo tras la mejora de uno de sus subsistemas. Si lo que se evalúa es un programa representa el cociente entre el tiempo de ejecución completo del programa antes de mejorar alguno de sus componentes y el tiempo de ejecución después de ser mejorado dicho componente.
- α_m: Representa el factor de mejora que se ha introducido en el subsistema alterado, nos mide *cómo de grande* ha sido la mejora. Es la aceleración parcial correspondiente al elemento mejorado considerado aisladamente.
- p: Es la fracción de tiempo que el sistema completo original utiliza el subsistema que se ha alterado (es decir, *cuánto se usa* el componente que se ha mejorado en el sistema original).

Podemos generalizar la ley de Amdahl cuando se realiza la optimización de más de una de las partes del sistema. Si n partes de un sistema consumen fracciones del tiempo total (secuencial) p_1, p_2, ... p_n respectivamente y cada una de ellas se optimiza individualmente con aceleraciones parciales α_1, α_2, ..., α_n, quedando sin optimizar por tanto la fracción $1-(p_1+p_2+...+p_n)$ del tiempo secuencial original, la aceleración global sufrida por el tiempo consumido por el sistema viene dado por:

$$\alpha = \frac{1}{\left(1 - \sum_{i=1}^{n} p_i\right) + \sum_{i=1}^{n} \frac{p_i}{\alpha_i}}$$

Se deja al lector la deducción de la expresión de la ley de Amdahl, a partir de la definición de los parámetros, así como su generalización.

B.1.1. Ejemplo

Queremos mejorar el rendimiento de un computador introduciendo un coprocesador matemático que realice las operaciones aritméticas en la mitad de tiempo. Calcular la ganancia en velocidad del sistema para la ejecución de

un programa sabiendo que el 60 % de dicha ejecución se dedica al cálculo de operaciones aritméticas. Si el programa tardaba 12 segundos en ejecutarse sin la mejora, ¿ Cuánto tardará en ejecutarse sobre el sistema mejorado?

SOLUCIÓN:

Del enunciado tenemos que $\alpha_m = 2$ y $p = 0.6$. Aplicando la ley de Amdahl, obtenemos:

$$\alpha = \frac{1}{(1-0.6) + \frac{0.6}{2}} = \frac{1}{0.4 + 0.3} = \frac{1}{0.7} = 1.42$$

Es decir, el sistema funciona un 42 % más rápido que el original.

Para responder a la segunda cuestión, expresamos α como el cociente entre los tiempos de ejecución sin y con mejora, esto es:

$$\alpha = \frac{\text{Tiempo de ejecución sin mejora}}{\text{Tiempo de ejecución con mejora}}$$

Y dado que el numerador de esta expresión es 12 segundos, sustituyendo A por 1.42 y despejando, obtenemos que el programa tardará 8.45 segundos cuando se ejecute aprovechando el coprocesador matemático.

B.1.2. Casos límite

De la fórmula de la ley de Amdahl podemos extraer unos cuantos casos particulares que merece la pena analizar ya que representan las limitaciones que tenemos a la hora de ampliar el rendimiento global del sistema, al mejorar únicamente una de sus partes.

1. Si $\alpha_m = \infty$, entonces $\alpha = \frac{1}{1-p}$: es decir, el porcentaje máximo que un sistema puede acelerarse actuando sobre uno de sus componentes está acotado en función de la fracción de uso de ese componente, como puede apreciarse en la figura B.1 (a).
2. Si $p = 0$, entonces $\alpha = 1$: la mejora de un componente, por grande que sea, no tiene efecto alguno sobre el sistema global si no se utiliza dicho componente, ver figura B.1 (b).
3. Si $p = 1$, entonces $\alpha = \alpha_m$: esto es, si todo el tiempo de ejecución de un programa se dedica a utilizar el subsistema mejorado, toda la ganancia en velocidad que experimente dicho subsistema revierte sobre el sistema general.

En definitiva, la ley de Amdahl representa una guía para conocer la mejor forma de repartir el dinero a invertir en un sistema, por ejemplo un equipo informático, entre los distintos componentes que lo integran en función de su

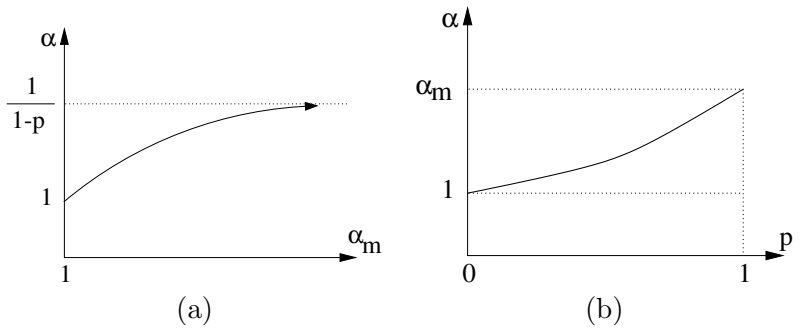


Figura B.1: Casos Límite de la Ley de Amdahl.

uso, y de cuantificar el efecto que estos componentes tendrán sobre el sistema en función de sus prestaciones.

B.2. OBSERVACIÓN DE MOORE

En 1965, Gordon Moore, cofundador de Intel, publicó un trabajo en el que exponía que había observado en los últimos años como el número de transistores por (pulgada)^2 de los circuitos integrados se habían duplicado cada año. Moore además predijo que esta tendencia continuaría previsiblemente en el futuro. En los años siguientes, la cosa se frenó un poco, pero la densidad de los datos se duplicó aproximadamente cada 18 meses, lo que constituye la actual definición de la *Ley de Moore*, que él mismo ha corroborado. La mayoría de expertos, incluido el propio Moore, esperan que la ley de Moore se mantenga por al menos otras dos décadas. En la figura B.2 se puede observar la ley de Moore para los procesadores de Intel.

Existe una versión de la ley de Moore para discos duros. Según sus previsiones, para el 2030 se llegarán a los 10^9 MBytes por pulgada cuadrada. Teniendo en cuenta el crecimiento esperado del tamaño del software, será precisa esta capacidad.

En relación con este aspecto, Nathan Myhrvold, un alto ejecutivo de Microsoft, formuló la conocida como *primera ley del software*, que establece que el software es como un gas: se expande hasta llenar el continente donde se encuentra (hardware).

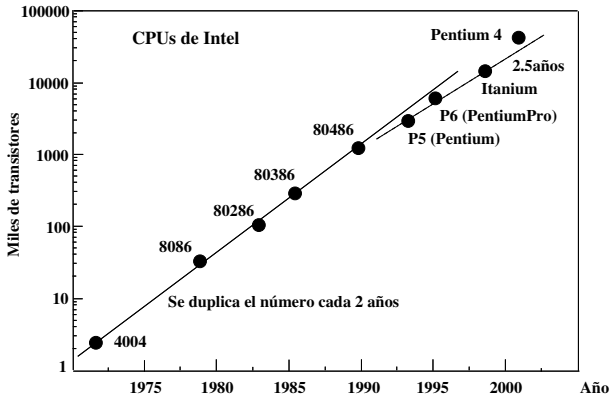


Figura B.2: Observación de Moore para las CPUs de Intel

B.3. OBSERVACIÓN DE GROSCH

La observación Grosch viene a decir que *la potencia de computación se incrementa como el cuadrado de el coste del computador* ó $P = KC^2$, donde P es la potencia de cálculo, K es una constate y C es el coste del sistema. Es decir, por el doble de dinero tenemos un ordenador 4 veces más potente.

La realidad nos dice que en el mercado de los PCs esto es solo una aproximación. Sin embargo, la ley de Grosch nos da una buena idea cuál será el precio de un PC más rápido en el futuro.

B.3.1. Ejemplo

Suponiendo que la potencia de computación de un PC se puede considerar proporcional a su frecuencia de reloj, estimar cuál será el precio que tendrá el Pentium a 2 GHz, teniendo en cuenta que un Pentium típico a 500 MHz tiene un precio de unas 1.000 euros.

Solución:

Aplicando la ley de Grosch, tenemos que el precio será:

$$1.000 \times \sqrt{\frac{2000}{500}} = 1.000 \times 2 = 2.000 \; euros$$

B.4. EJERCICIOS

1. Queremos analizar el rendimiento de un computador de 100 MHz. que tiene instrucciones simples, normales y complejas, cada una con un CPI distinto según se muestra en la tabla. Para ello, se utiliza un programa compilado en dos compiladores distintos que generan códigos objeto con diferente número y tipo de instrucciones, (ver tabla). ¿ Qué versión compilada del programa produce un mayor rendimiento atendiendo a los MIPS? ¿Y atendiendo al tiempo de CPU?

Tipo de instrucción	CPI	Millones de instrucciones	
		Compilador 1	Compilador 2
Simple	1	5	10
Normal	2	1	1
Compleja	3	1	1

2. Considerar dos computadores, P1 y P2, con idéntico repertorio de instrucciones (A, B, C, D). P1 funciona a 50 MHz y con CPIs 1, 2, 3 y 4 para A, B, C y D, respectivamente. P2, por el contrario, funciona a 75 MHz. pero con CPIs respectivos de 3, 5, 5 y 7. Si ejecutamos en ambos un mismo programa compuesto por igual número de instrucciones de cada tipo, ¿Cuál de ellos mostrará un mayor rendimiento en MIPS?
3. *MeEscapé*, una empresa dedicada al software de Internet, usa, para sus operaciones matemáticas, rutinas compradas a otra compañía. Estas funciones suponen el 10 % del tiempo de ejecución del código, y son el doble de rápidas de las que inicialmente usaban en la primera versión de sus productos.

   Por una política de ahorro, se decide cambiar a las funciones que ofrece una nueva empresa, que aunque son algo más lentas, también son más baratas. Esta decisión se debe a que se ha observado que haciendo un cambio en la estructura de los programas, se consigue que, aunque las nuevas llamadas sean un 25 % más lentas (que las que iban el doble de rápidas), el código, de forma global, siga comportándose de igual modo a como lo hacía antes en cuanto a velocidad de ejecución.

   a) ¿ Cuál ha sido ese cambio? ¿ En qué se basa ese comportamiento, en principio extraño, del programa? Justifíquese la respuesta.
   b) Cuantificar el cambio necesario en el programa, como mínimo, para que se puedan usar las nuevas funciones.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

4. El nuevo procesador de la compañía *Acme Microsys* se encuentra actualmente en fase de desarrollo. Los únicos datos que se conocen son relativos un programa multimedia de prueba, el cual tarda 15 segundos en ejecutarse, con un CPI medio de 3, y en el que cada tipo de instrucción gráfica necesita una media de 4 ciclos de reloj para ejecutarse utilizando la tarjeta de video T1. Una prueba posterior del mismo programa sobre otra tarjeta (T2) redujo a la mitad los ciclos empleados por cada tipo de instrucción gráfica, lo que implicó una reducción de 5 segundos en el tiempo de ejecución de la prueba.
   a) Calcular el CPI medio de la segunda prueba.
   b) Sabiendo que la prueba tiene un total de 5000 instrucciones, calcula el número de instrucciones gráficas contenidas en dicho programa.
   c) Con la nueva tarjeta de vídeo T3, el CPI de la prueba pasó a ser 1.8. Obtener el número de ciclos medio que debe tardar cada instrucción gráfica con la nueva tarjeta T3.
5. Un computador ejecuta un programa en 100 segundos, siendo las operaciones de multiplicación responsables del 80 % de ese tiempo. ¿ Cuánto habría que mejorar la velocidad de la multiplicación si se desea que el programa se ejecute 5 veces más rápido? *Solución: No hay forma de conseguir una mejora de tal magnitud para ese programa acelerando sólo la multiplicación.*
6. Un programa que dedica la mitad de su tiempo a cálculos en punto flotante se ejecuta sobre un computador en 10 segundos. Si cambiamos su FPU por otra 5 veces más rápida, ¿Qué ganancia en velocidad experimentará el programa? *Solución: $A = 1.6$*
7. Queremos reducir el tiempo de ejecución de un programa en un computador incorporándole una memoria caché para almacenar instrucciones. Cuando el procesador encuentra una instrucción en la caché, ésta se ejecuta 10 veces más rápido que cuando debe acceder a memoria principal. Indicar el porcentaje mínimo de instrucciones que se deben encontrar en caché para conseguir ejecutar el programa en menos de la mitad de tiempo. *Solución: 55 %*
8. Una forma de mejorar el rendimiento de un computador es utilizando un sistema multiprocesador. Generalmente, sólo una parte del programa a ejecutar puede *paralelizarse*, esto es, admite una descomposición en tareas que permita beneficiarse de la existencia de múltiples CPUs. Si queremos medir el rendimiento de una máquina de N procesadores a través de un benchmark que tarda 100 segundos en una máquina se-

cuencial y donde el 90 % de sus tareas son paralelizables, se pide:

a) La aceleración que experimentará el benchmark en este sistema en función del número de procesadores (N).

b) Si pudiésemos incrementar el número de procesadores de forma indefinida, ¿ Estarían acotados de alguna manera la aceleración y el tiempo de ejecución para esta máquina?

Solución: $a = \frac{100}{10+\frac{90}{N}}$. *Por mucho que aumentemos N, la aceleración se encuentra acotada superiormente en un valor 10 y el tiempo de ejecución inferiormente en 10 segundos.*

9. Se desea mejorar el rendimiento de un PC. Para ello tenemos dos opciones:

a) Ampliar la memoria RAM, con lo que se consigue que el 80 % de los programas vayan 1.75 veces más rápido.

b) Introducir un disco duro mayor, con lo que el 60 % de los programas realizan su tarea en la tercera parte del tiempo original.

A igualdad de precio, ¿ Cuál de las dos mejoras aconsejarías ? *Solución: La primera alternativa produce una aceleración de 1.52, mientras que la segunda 1.66. Nos quedamos pues con la mejora del disco duro.*

Apéndice C

Códigos completos para el i8086

C.1. CÓDIGOS COMPLETOS

Los siguientes códigos corresponden con algunos de los problemas del capítulo 3. En este apéndice los códigos se muestran de forma completa, incluyendo los segmentos de datos, pila y código para poder ser ensamblados.

1. Tenemos 250 números en C2 de 16 bits en las posiciones 500 a 998 de memoria. Escribir un programa en ensamblador que cuente los números $P \geq 0$ y $N < 0$ y que almacene P en la posición 1000 de memoria y N en la 1002.

```
;----------------
DATOS  SEGMENT
       ; aqui declarar variables si hace falta
DATOS  ENDS
;----------------
PILA   SEGMENT
       DB 512 DUP(0)
PILA   ENDS
;----------------
CODIGO SEGMENT
       ASSUME CS:CODIGO, DS:DATOS, SS:PILA
INICIO:MOV CX, 0H   ; contador de positivos o cero
       MOV DX, 0H  ; contador de negativos
       MOV DS, 0H  ; los datos el el segmento 0
       MOV SI, 500 ; inicializar desplazamiento
;----------------
```


R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

```
 LAZO: CMP SI, 998 ; hemos llegado al final?
       JG FIN
       CMP WORD PTR [SI],0H ; Direcionamos DS:SI como palabra
       JLE NEGA
       ADD CX,1
       ADD SI,2    ; SI apunta a la siguiente palabra
       JMP LAZO
 NEGA: ADD DX,1
       ADD SI,2    ; SI apunta a la siguiente palabra
       JMP LAZO
;ahora SI=1000
  FIN: MOV [SI], CX
       MOV [SI+2], DX
;----------------
  FINP:MOV AH,4CH  ; interrupcion fin de programa
       INT 21H
CODIGO ENDS
;----------------
       END INICIO
```

2. Tenemos 100 números positivos de 16 bits en las posiciones de memoria 1000 a 1198. Escribir un programa que determine cuál es el número mayor de todos y lo almacene en la posición 1200.

```
;----------------
DATOS  SEGMENT AT 1200 ; variable que contiene el mayor en 1200
       MAYOR DW 0
DATOS  ENDS
;----------------
PILA   SEGMENT STACK
       DB 512 DUP(0)
PILA   ENDS
;----------------
CODIGO SEGMENT
       ASSUME CS:CODIGO, DS:DATOS, SS:PILA
INICIO:MOV ES, 0H  ; los datos el el segmento 0
       MOV SI, 1000; inicializar desplazamiento
       MOV AX, SEG DATOS
       MOV DS, AX  ; iniciar DS apuntando a datos
;----------------
 LAZO: CMP SI, 1198;  hemos llegado al final?
       JG FIN
       CMP WORD PTR ES:[SI], WORD PTR MAYOR
       JLE MENOR
       MOV MAYOR, WORD PTR ES:[SI]
 MENOR:
```

```
       ADD SI,2    ; SI apunta a la siguiente palabra
       JMP LAZO
;---------------
  FIN: MOV AH,4CH  ; interrupcion fin de programa
       INT 21H
CODIGO ENDS
;---------------
       END INICIO
```

3. Escribir un procedimiento en lenguaje ensamblador para calcular el sumatorio de los números naturales desde 1 hasta N. Supongamos que N está almacenado en la posicion fisica 500 y queremos almacenar el sumatorio en la posición 502.

```
;---------------
DATOS  SEGMENT AT 500
       NUMERO DW 6
       SUMAT  DW ?
DATOS  ENDS
;---------------
PILA   SEGMENT  STACK
       DB 512 DUP(0)
PILA   ENDS
;---------------
CODIGO SEGMENT
       ASSUME CS:CODIGO, DS:DATOS, SS:PILA
       MOV AX, SEG DATOS
       MOV DS, AX  ; DS apuntando a datos
;---------------
INICIO: MOV AX, NUMERO; guardo el numero en AX
       MOV CX, 0     ; inicio CX donde metere el sumatorio
 LAZO: ADD CX,AX
       SUB AX,1
       CMP AX,0
       JNE LAZO
  FIN: MOV SUMAT, CX
;---------------
  FINP:MOV AH,4CH  ; interrupcion fin de programa
       INT 21H
CODIGO ENDS
;---------------
       END INICIO
```

4. Escribe el diagrama de flujo y el programa en ensamblador del para el algoritmo que realiza la siguiente función: Dada una tabla en la posi-

ción 100 de memoria con 50 números de 16 bits en complemento a dos, multiplicar por 2 los números pares y dividir por 2 los impares (división entera), dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar las instrucciones de multiplicación y división (MULS y DIVS). No considerar el caso de *overflow* al multiplicar por dos.

```
;----------------
DATOS  SEGMENT AT 100
       TABLA DW 50 DUP(?)   ; zona de memoria con los numeros
DATOS  ENDS
;----------------
PILA   SEGMENT STACK
       DB 512 DUP(0)
PILA   ENDS
;----------------
CODIGO SEGMENT
       ASSUME CS:CODIGO, DS:DATOS, SS:PILA
       MOV AX, SEG DATOS
       MOV DS, AX           ;cargar DS con el seg de datos
       LEA SI, TABLA        ;SI apuntando a TABLA
;----------------
       MOV DX,50            ;contador de elementos de la tabla
LAZO:  MOV AX, [SI]
       MOV BX, AX
       AND BX, 01H
       CMP BX, 01H
       JE  IMPAR
       SAL AX, 01H
       JMP SEGUIR
IMPAR: SAR AX, 01H
SEGUIR:MOV [SI], AX
       ADD SI, 2   ; SI apunta a la siguiente palabra
       SUB DX, 1   ; decremento contador del bucle
       CMP DX, 0
       JNE LAZO
;----------------
  FINP:MOV AH,4CH  ; interrupcion fin de programa
       INT 21H
CODIGO ENDS
;----------------
       END INICIO
```

5. Escribe el diagrama de flujo y el programa en ensamblador para el algoritmo que realiza la siguiente función: Dada una tabla en la posición

500 de memoria con 50 números de 16 bits en complemento a dos, forzar a cero los números negativos y multiplicar por tres los positivos, dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar ninguna instrucción de multiplicar (MULS). No considerar el caso de *overflow* al multiplicar por tres. (Observar que multiplicar por 3 es sumar a un numero su doble).

```
;----------------
DATOS  SEGMENT AT 100
       TABLA DW 50 DUP(?)   ; zona de memoria con los numeros
DATOS  ENDS
;----------------
PILA   SEGMENT STACK
       DB 512 DUP(0)
PILA   ENDS
;----------------
CODIGO SEGMENT
       ASSUME CS:CODIGO, DS:DATOS, SS:PILA
       MOV AX, SEG DATOS
       MOV DS, AX              ;cargar DS con el seg de datos
       LEA SI, TABLA           ;SI apuntando a TABLA
;----------------
       MOV DX,50
LAZO:  MOV AX, [SI]
       CMP AX, 00H
       JE  NEGA
       SAL AX, 01H
       ADD [SI], AX
       JMP SEGUIR
NEGA:  MOV [SI], 0H
SEGUIR:ADD SI, 2               ;apuntar a la siguiente palabra
       SUB DX, 1
       CMP DX, 0
       JNE LAZO
;----------------
  FINP:MOV AH,4CH  ; interrupcion fin de programa
       INT 21H
CODIGO ENDS
;----------------
       END INICIO
```

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Apéndice D

Detalles de diseño de cache

D.1. POLÍTICAS DE ASIGNACIÓN

Veamos en más detalle como se mapean los datos de la memoria principal en la memoria caché. Vamos a considerar una memoria principal de 2^N líneas, una memoria caché de 2^n líneas de capacidad y 2^w palabras por línea.

D.1.1. Asignación directa

Con este criterio, la correspondencia entre memoria principal y caché se hace de la siguiente manera: la línea i de la memoria principal se ubica en la línea $i \, mod \, 2^n$ de la caché. Así, los subconjuntos de líneas de memoria principal que pueden ser almacenadas en cada línea caché son disjuntos.

Partiendo de la dirección de palabra de memoria principal, es fácil ubicar la línea correspondiente en memoria: los últimos w bits indican *qué palabra referenciamos dentro de la línea*, los n siguientes indican *qué linea referenciamos dentro de la caché* y el resto (los más significativos) nos sirven para determinar si la línea está en caché o no (etiqueta) (ver figura D.1).

Entre las ventajas de este tipo de asignación encontramos su simplicidad y su bajo coste: sólo hace falta el uso de un comparador para saber si la línea está en la caché, el directorio caché es una memoria convencional (no asociativa) y no hace falta algoritmo de reemplazo (cada línea de memoria principal se mapea de forma unívoca en la caché).

Como inconveniente se encuentra el hecho de que la caché puede tener líneas vacías y sin embargo, ocurra reemplazo (cuando los accesos consecutivos a memoria principal están a una distancia 2^n líneas).

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

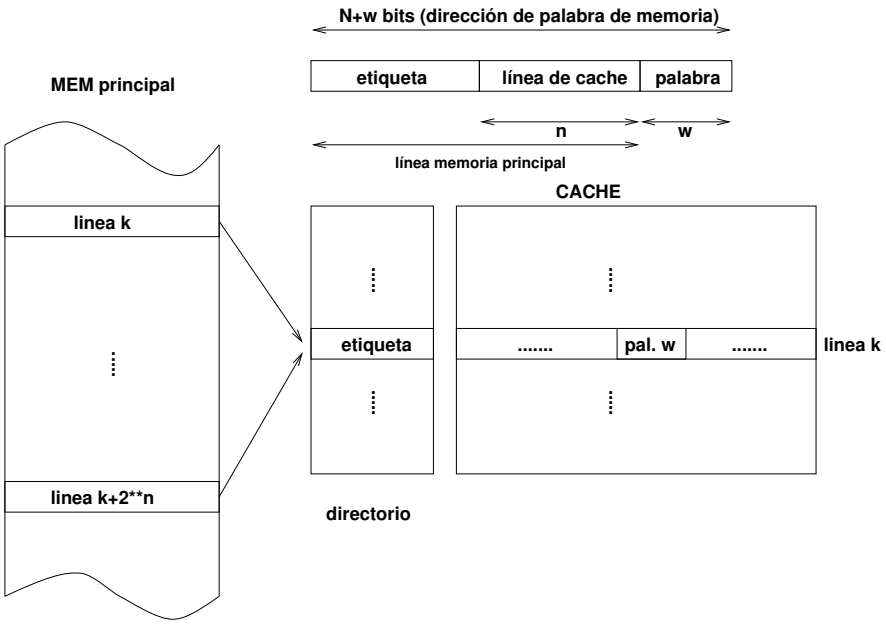


Figura D.1: Asignación directa

D.1.2. Asignación completamente asociativa

Para paliar el inconveniente anterior, una organización totalmente asociativa va a permitir que cualquier línea de memoria principal pueda ubicarse en cualquier línea de la caché. Para ello, partiendo de la dirección de la palabra referenciada, los N bits más significativos indican *a qué línea pertenece la palabra* en memoria principal. Por tanto, guardando esta información en el directorio, y realizando una búsqueda asociativa en él, determinamos si se encuentra o no la línea en caché y, en caso de que se encuentre, cuál es su posición en la caché (ver figura D.2).

Vemos que este criterio maximiza la flexibilidad y el rendimiento al poder ser ocupada cualquier posición libre de la caché. Como contrapartida el coste aumenta: se necesita una memoria asociativa y además es necesario implementar un algoritmo de reemplazo.

D.1.3. Asignación asociativa por conjuntos

Es una organización de compromiso entre las dos anteriores. La caché se organiza en 2^c unidades que denominaremos conjuntos, cada uno de los cuales va a

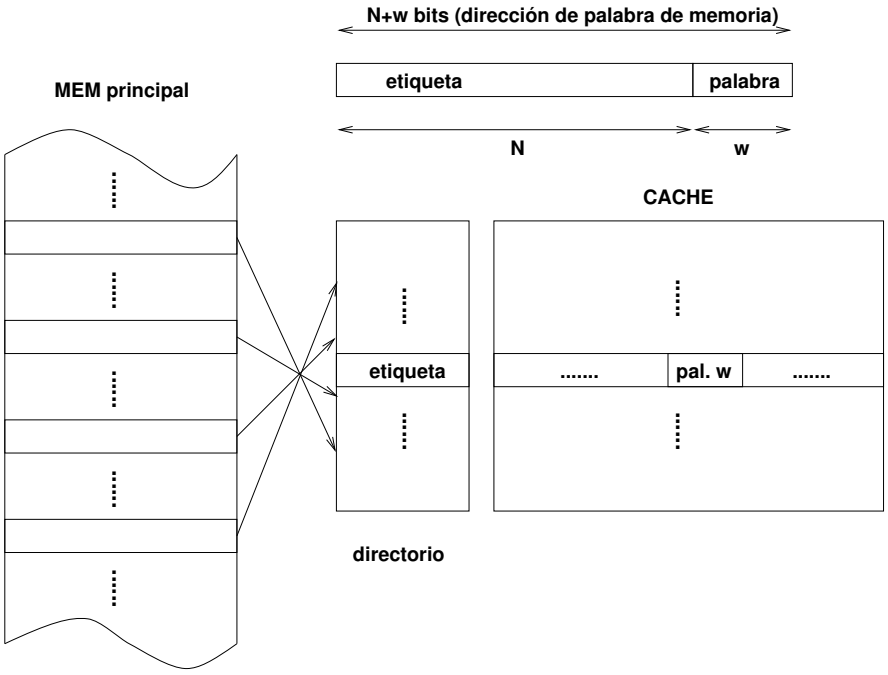


Figura D.2: Asignación completamente asociativa

contener 2^{n-c} líneas. Al número de líneas por conjunto le denominaremos *nivel de asociatividad*. Lo que se pretende hacer es que la selección del conjunto se realice con mapeo directo y, una vez seleccionado el conjunto, la ubicación dentro de éste sea asociativa.

Para ello, la línea de memoria principal i será cargada en caché en el conjunto $i \, mod \, 2^c$ y, dentro de ese conjunto, podrá ocupar cualquier línea caché. Así, una dirección de palabra de la memoria principal la dividiremos en tres campos: los w bits menos significativos indican como antes *qué palabra referenciamos dentro de la línea*, los c bits siguientes seleccionan *a qué conjunto de caché va destinada* y los $N - c$ bits más significativos constituyen la etiqueta a guardar para la búsqueda asociativa dentro del conjunto que le haya correspondido (ver figura D.3). Observemos que cuando un conjunto está lleno hay que aplicar alguna política de reemplazo para sustituir las líneas dentro de ese conjunto.

Observemos que cuando $c = 0$, es decir, sólo hay un conjunto, la organización equivale a una totalmente asociativa, y que cuando $c = n$, es decir, 2^n conjuntos (nivel de asociatividad $= 1$), la organización equivale a la de

asignación directa.

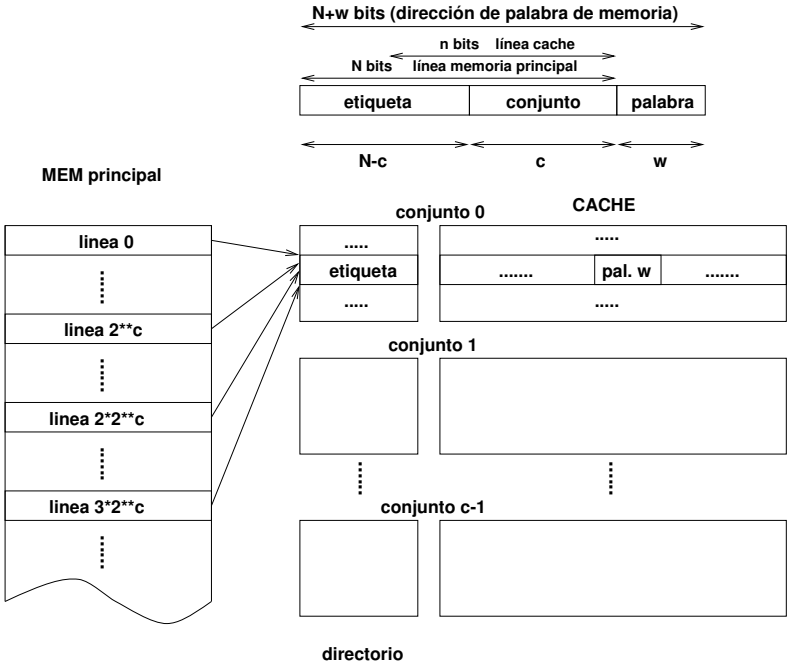


Figura D.3: Asignación asociativa por conjuntos

D.2. REEMPLAZO Y ACTUALIZACIÓN

Como ya se ha esbozado en párrafos anteriores, en las organizaciones con más de una línea por conjunto, es necesario introducir una **política de reemplazo** de aquellas líneas que son referenciadas en memoria principal y que pueden ser ubicadas en diferentes posiciones dentro de la caché. Los algoritmos de reemplazo se diseñan con el objetivo de disminuir la tasa de fallos, minimizando el tráfico entre memoria principal y caché. Estos algoritmos, además, deben estar implementados en hardware para no degradar la velocidad de respuesta del sistema.

Los tres algoritmos más usados son:

1. *LRU (Least Recently Used)*: Se elimina aquella línea que hace mass tiempo que no se usa (lectura o escritura).
2. *FIFO (First In First Out)*: Se elimina aquella línea que entró en la caché hace más tiempo.

3. *RANDOM*: Se escoge la línea a eliminar aleatoriamente.

El menos complejo de implementar es el último, pero sus resultados son peores, ya que no utiliza ninguna información relacionada con la localidad. FIFO se puede implementar simplemente asociando a cada conjunto un contador módulo el nivel de asociatividad, que apuntaría a la siguiente línea a sustituir. LRU es más complejo de implementar (hay que medir de alguna forma cuánto lleva la línea en caché), aunque también es el que presenta mejores resultados.

Con **Algoritmo de Carga** hacemos referencia a la manera de decidir qué información se carga en la caché y cuándo. Lo más simple es realizar la carga *bajo demanda*, esto es, cargar en caché aquella línea de memoria principal a la que pertenece la palabra de memoria referenciada. Otra posibilidad sería usar un algoritmo de *precarga*, en el que no sólo cargaremos en caché la línea referenciada, sino otras próximas a ésta (por ejemplo algunas consecutivas). En este caso, son diferentes los factores a tener en cuenta: cuántas y cuáles líneas son las que vamos a precargar, cuándo se realiza la precarga (siempre, bajo fallo, ...), etc.

La información de la memoria principal debe ser coherente con la que se encuentra en la caché. Esta información se modifica cuando se efectúa una escritura en la memoria. Encontramos dos políticas básicas de **actualización de memoria**:

1. *Escritura directa o Write-through*: Cada vez que se actualiza (escribe) una posición en la caché es actualizada su posición correspondiente en la memoria principal. Con esto garantizamos una buena consistencia de los datos, mientras que, por otra parte, el tráfico entre caché y memoria principal aumenta.
2. *Post-escritura o Write-back*: Los datos se escriben en la caché y el nuevo dato no pasa a memoria principal hasta cierto tiempo después (por ejemplo, cuando la línea modificada es eliminada de la caché). En este caso, habrá que añadir un bit al directorio indicando que la línea ha sido modificada. El tráfico se reduce, pero la consistencia de datos podría dar problemas en caso de sistemas multiprocesadores o DMA (Direct Memory Access).

Otro factor de diseño es el empleo de **cachés separadas**, una para las referencias a datos y otra para las referencias a instrucciones. Con ello conseguimos aumentar el ancho de banda, ya que se pueden acceder a las dos simultáneamente (por ejemplo, con un procesador segmentado) y disminuir el tiempo de acceso si físicamente situamos la caché de datos cerca de la sección de procesamiento y la de instrucciones cerca de la lógica de decodificación.

Por último, apuntar que es posible interponer **varios niveles** de caché entre la memoria principal y el microprocesador. Cada caché trataría la de nivel superior como si fuera su memoria principal, funcionando tal como hemos visto, siendo cada nivel más rápido y de menor capacidad cuanto más cerca se encuentre del microprocesador.

D.3. RENDIMIENTO DE MEMORIAS CACHÉS

El rendimiento de un sistema caché vendrá dado por la relación existente entre los fallos y aciertos y la latencia de los diferentes elementos que componen el sistema.

Para una traza dada la relación entre fallos, aciertos y precarga se suele medir mediante los siguientes parámetros:

Índice o tasa de fallos: Es el cociente $P_{miss} = \frac{\text{N.º fallos}}{\text{N.º referencias}}$ y representa la probabilidad de que una petición genere un fallo de caché.

Índice o tasa de aciertos: Es el cociente $P_{success} = \frac{\text{N.º aciertos}}{\text{N.º referencias}}$ y representa la probabilidad de que una petición genere un éxito de caché. Se tendrá que $P_{success} + P_{miss} = 1$.

Índice o tasa de precarga: Es el cociente $P_{prefetch} = \frac{\text{N.º precargas}}{\text{N.º referencias}}$ y representa la probabilidad de que una petición genere una precarga.

A continuación, a modo de ejemplo, se evalúa el rendimiento de diferentes topologías determinando el tiempo de acceso equivalente del sistema de memoria.

Topología caché de un solo nivel unificada sin precarga. Usaremos un modelo caracterizado por los siguientes parámetros: t_c tiempo de latencia de la caché, T_{rmc} tiempo de transferencia o transporte de memoria principal a caché.

El tiempo medio consumido nos proporciona la latencia efectiva del sistema completo y viene dado por la expresión:

$$T = T_{success} + T_{miss} = (1 - P_{miss})t_c + P_{miss}(t_c + T_{rmc})$$

Si consideramos que el tiempo de acceso a memoria principal sin caché es aproximadamente T_{rmc}, el *speed-up* obtenido gracias al sistema caché vendrá dado por:

$$\alpha = \frac{T_{no\ cache}}{T_{cache}} = \frac{T_{rmc}}{(1 - P_{miss})t_c + P_{miss}(t_c + T_{rmc})}$$

Topología caché de un solo nivel unificada con precarga. Usando el mismo modelo de antes, y considerando t_p la penalización introducida por cada precarga, basta sumar el tiempo adicional que representan éstas, para calcular la latencia equivalente:

$$T = T_{success} + T_{miss} + T_{prefetch} = (1 - P_{miss})t_c + P_{miss}(t_c + T_{rmc}) + P_{prefetch}\, t_p$$

La precarga será beneficiosa si este tiempo es menor al que se obtendría en caso de que la precarga no estruviera presente. Observemos que el índice de fallos y aciertos se modifica al introducir un algoritmo de precarga y que en condiciones de localidad favorables es de esperar que el aumento del índice de aciertos compense la penalización de las líneas precargadas.

Topología caché unificada de dos niveles sin precarga. Denominando t_{c1} el tiempo de latencia de la caché de primer nivel (L1), t_{c2} el tiempo de transporte de los datos desde el primer (L1) al segundo nivel (L2) y T_{rmc} tiempo de transferencia o transporte de memoria principal al segundo nivel y considerando una tasa de fallos en el primer nivel de P_{miss_1} y una tasa de fallos en el segundo nivel de P_{miss_2}, la latencia efectiva, viene dada para este caso por:

$$T = T_{success} + T_{miss\,L1} + T_{miss\,L2} = (1 - P_{miss_1})t_{c1} + P_{miss_1}(1 - P_{miss_2})(t_{c1} + t_{c2}) + P_{miss_1}P_{miss_2}(t_{c1} + t_{c2} + T_{rmc})$$

Topología caché separada Datos+Instrucciones. El rendimiento se determina de forma análoga al caso unificado, sólo que considerando separadamente, en la traza de peticiones, lo que son peticiones de datos y peticiones de instrucciones. Si T_{data} es la latencia equivalente en la caché de datos, $T_{\text{i's}}$ la de la caché de instrucciones, P_{data} la probabilidad de petición de datos en la traza completa y $P_{\text{i's}}$ la probabilidad de petición de instrucciones, la latencia efectiva se determinará como:

$$T = P_{\text{data}}T_{\text{data}} + P_{\text{i's}}T_{\text{i's}}$$

D.4. EJERCICIOS

1. Dado un sistema de memoria principal de 1 Mb. y un programa que referencia a la siguiente secuencia de direcciones (en hexadecimal):

   `ABC80h, ABC81h, ABC88h, BCD90h, BCD9Dh, BCDA0h,`
   `CDE00h, CDE18h, CDE20h`

   Considerar las dos estrategias que se detallan a continuación para acelerar el acceso a memoria principal:

   I) Entrelazar la memoria principal siguiendo un esquema de orden inferior con 32 módulos y latches a la salida.

   II) Incorporar una memoria caché de 64 Kb. organizada asociativamente en 4 conjuntos, 8 palabras por linea, sin precarga y algoritmo de reemplazo FIFO.

   Se pide:

   a) Analizar y comparar el tiempo total de respuesta (en ciclos de CPU) de esos dos sistemas de memoria para la secuencia de direcciones solicitada por el programa. Considerar un tiempo de acceso de 10 ciclos de CPU para la memoria principal y de un solo ciclo para la caché y los latches de la memoria entrelazada. Suponer la caché inicialmente vacía e ilustrar claramente el contenido de los latches y del directorio caché tras cada acceso a memoria principal.

   b) Discutir posibles mejoras en ambos sistemas y su repercusión en el coste: En el entrelazado, respecto al número de módulos y la colocación de los latches; en la caché, respecto a su organización y a los algoritmos para precarga y reemplazo de líneas.

   SOLUCIÓN:

   Como el factor de entrelazamiento es 32, los últimos 5 bits determinan el módulo en el que se encuentra una referencia. Así si se hace referencia a una posición A, se lee a la vez desde la posición $(A\,div\,32)*32$ hasta $(A\,div\,32)*32+31$, siendo div la división entera. (La dirección seleccionada en el módulo 0 se obtiene poniendo los últimos 5 bits de A a 0, y la dirección en el módulo 31 poniendo los últimos 5 bits a 1).

```
referencia     palabra en latches?          tiempo
ABC80h ->      NO. Cargar ABC80...ABC9F     10+1 ciclos
ABC81h ->      SI.                           1 ciclo
ABC88h ->      SI.                           1 ciclo
BCD90h ->      NO. Cargar BCD80...BCD9F     10+1 ciclos
BCD9Dh ->      SI                            1 ciclo
BCDA0h ->      NO. Cargar BCDA0...BCDBF     10+1 ciclos
CDE00h ->      NO. Cargar CDE00...CDE1F     10+1 ciclos
CDE18h ->      SI                            1 ciclo
CDE20h ->      NO. Cargar CDE20...CDE3F     10+1 ciclos
```

El sistema de caché propuesto evoluciona de la siguiente manera:

```
Parametros:
 Memoria principal  = 1Mpalabras->N+w=20
 Memoria caché      = 8Klineas  ->n=13
 Palabras por línea = 8         ->w=3
 Numero de conjuntos= 4         ->c=2

Lineas en cache:8192, Conjuntos:4,
Lineas por conjunto:2048,
Reemplazo:FIFO Precarga:NO

Ref. 0xabc80, etiqueta=0x55e4,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     0
Ref. 0xabc81, etiqueta=0x55e4,
              c=0x0, w=0x1
   ----Exito
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
Ref. 0xabc88, etiqueta=0x55e4,
              c=0x1, w=0x0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x1      0x0      0x55e4     2
Ref. 0xbcd90, etiqueta=0x5e6c,
              c=0x2, w=0x0
   ----Linea vacia: conjunto:0x2,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
Ref. 0xbcd9d, etiqueta=0x5e6c,
              c=0x3, w=0x5
   ----Linea vacia: conjunto:0x3,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
    0x3      0x0      0x5e6c     4
Ref. 0xbcda0, etiqueta=0x5e6d,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x1
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x0      0x1      0x5e6d     5
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
    0x3      0x0      0x5e6c     4
Ref. 0xcde00, etiqueta=0x66f0,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x2
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x0      0x1      0x5e6d     5
    0x0      0x2      0x66f0     6
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
    0x3      0x0      0x5e6c     4
Ref. 0xcde18, etiqueta=0x66f0,
              c=0x3, w=0x0
   ----Linea vacia: conjunto:0x3,
                    linea en conj.:0x1
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x0      0x1      0x5e6d     5
    0x0      0x2      0x66f0     6
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
    0x3      0x0      0x5e6c     4
    0x3      0x1      0x66f0     7
Ref. 0xcde20, etiqueta=0x66f1,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x3
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x55e4     1
    0x0      0x1      0x5e6d     5
    0x0      0x2      0x66f0     6
    0x0      0x3      0x66f1     8
    0x1      0x0      0x55e4     2
    0x2      0x0      0x5e6c     3
    0x3      0x0      0x5e6c     4
    0x3      0x1      0x66f0     7

 Estadisticas:
  Referencias:                    9
  Lineas precargadas:             0
  Exitos en referencias:          1
  Exitos incluyendo precargaas: 1

Tiempo=num_fallos*(10+1ciclos)
       +num_aciertos*1ciclo
```

2. Considerar una memoria principal de 1 Mb. estructurada en palabras de 1 byte y a la que se dota de una memoria caché de 64 Kb., líneas de 4096 palabras, sin precarga y algoritmo de reemplazo FIFO.

   Sea la secuencia de acceso a memoria principal dada por las siguientes direcciones:

   `00000h, 01000h, 01001h, 0F000h, 10000h, 00000h, 40000h, 20000h, 080000h`

Para una organización directa, totalmente asociativa, y asociativa de 4 conjuntos, se pide:

a) Mostrar la evolución de la caché y su directorio en cada caso, indicando los fallos y precargas que se producen.

b) Calcular el coste de cada implementación con respecto al número de bits del directorio caché y al número y tamaño de las memorias asociativas que lo componen.

c) Ordenar las tres alternativas en base a criterios de rendimiento y coste.

Solución:

```
*ASIGNACION DIRECTA:
Param.: n= 4, c= 4, w=12
Cache de mapeo directo
Lineas en cache: 16
Palabras por linea: 4096
Conjuntos: 16
Lineas por conjunto: 1
No precarga

Ref. 0x0, etiqueta=0x0,
          c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
Ref. 0x1000, etiqueta=0x0,
             c=0x1, w=0x0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.   Ref.
    0x0     0x0     0
    0x1     0x0     1
Ref. 0x1001, etiqueta=0x0,
             c=0x1, w=0x1
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x0     2
Ref. 0xf000, etiqueta=0x0,
             c=0xf, w=0x0
   ----Linea vacia: conjunto:0xf,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x0     2
    0xf     0x0     3
Ref. 0x10000, etiqueta=0x1,
              c=0x0, w=0x0
   ----Reemplazo:  linea:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x1     4
    0x1     0x0     2
    0xf     0x0     3
Ref. 0x0, etiqueta=0x0,
          c=0x0, w=0x0
   ----Reemplazo:  linea:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     5
    0x1     0x0     2
    0xf     0x0     3
Ref. 0x40000, etiqueta=0x4,
              c=0x0, w=0x0
   ----Reemplazo:  linea:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     6
    0x1     0x0     2
    0xf     0x0     3
Ref. 0x20000, etiqueta=0x2,
              c=0x0, w=0x0
   ----Reemplazo:  linea:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     7
    0x1     0x0     2
    0xf     0x0     3
Ref. 0x80000, etiqueta=0x8,
              c=0x0, w=0x0
   ----Reemplazo:  linea:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x8     8
    0x1     0x0     2
    0xf     0x0     3
Estadisticas:
 Referencias:                              9
 Lineas precargadas:                       0
 Exitos en referencias:                    1
 Exitos incluyendo lineas precargadas:     1
 Fallos incluyendo lineas precargadas:     8
 Fallos compulsorios total:                3
```

```
*COMPLETAMENTE ASOCIATIVA:
Param.: n= 4, c= 0, w=12
Lineas en cache: 16
Palabras por linea: 4096
Conjuntos: 1
Lineas por conjunto: 16

Reemplazo: FIFO
No precarga.

Ref. 0x0, etiqueta=0x0,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
```

```
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
Ref. 0x1000, etiqueta=0x1,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x1
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x1     1
Ref. 0x1001, etiqueta=0x1,  w=0x1
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x1     2
Ref. 0xf000, etiqueta=0xf,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x2
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x1     2
    0x2     0xf     3
Ref. 0x10000, etiqueta=0x10,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x3
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     0
    0x1     0x1     2
    0x2     0xf     3
    0x3     0x10     4
Ref. 0x0, etiqueta=0x0,  w=0x0
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     5
    0x1     0x1     2
    0x2     0xf     3
    0x3     0x10     4
Ref. 0x40000, etiqueta=0x40,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x4
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     5
    0x1     0x1     2
    0x2     0xf     3
    0x3     0x10     4
    0x4     0x40     6
Ref. 0x20000, etiqueta=0x20,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x5
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     5
    0x1     0x1     2
    0x2     0xf     3
    0x3     0x10     4
    0x4     0x40     6
    0x5     0x20     7
Ref. 0x80000, etiqueta=0x80,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x6
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x0     5
    0x1     0x1     2
    0x2     0xf     3
    0x3     0x10     4
    0x4     0x40     6
    0x5     0x20     7
    0x6     0x80     8
 Estadisticas:
  Referencias:                              9
  Lineas precargadas:                       0
  Exitos en referencias:                    2
  Exitos incluyendo lineas precargadas:     2
  Fallos incluyendo lineas precargadas:     7
  Fallos compulsorios total:                7
```

```
*ASOCIATIVA CON 4 CONJUNTOS:
Param.: n= 4, c= 2, w=12
Lineas en cache: 16
Palabras por linea: 4096
Conjuntos: 4
Lineas por conjunto: 4
Reemplazo:FIFO
No precarga.

Ref. 0x0, etiqueta=0x0,
          c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.  Ref.
    0x0     0x0     0x0     0
Ref. 0x1000, etiqueta=0x0,
             c=0x1, w=0x0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0     0x0     0x0     0
    0x1     0x0     0x0     1
Ref. 0x1001, etiqueta=0x0,
             c=0x1, w=0x1
   ----Exito
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0     0x0     0x0     0
    0x1     0x0     0x0     2
Ref. 0xf000, etiqueta=0x3,
             c=0x3, w=0x0
   ----Linea vacia: conjunto:0x3,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0     0x0     0x0     0
    0x1     0x0     0x0     2
    0x3     0x0     0x3     3
Ref. 0x10000, etiqueta=0x4,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x1
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0     0x0     0x0     0
    0x0     0x1     0x4     4
    0x1     0x0     0x0     2
    0x3     0x0     0x3     3
Ref. 0x0, etiqueta=0x0, c=0x0, w=0x0
   ----Exito
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0     0x0     0x0     5
    0x0     0x1     0x4     4
    0x1     0x0     0x0     2
    0x3     0x0     0x3     3
```

```
Ref. 0x40000, etiqueta=0x10,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x2
   ----Directorio:
    Cto.     Lin.    Etq.   Ref.
    0x0      0x0     0x0     5
    0x0      0x1     0x4     4
    0x0      0x2     0x10    6
    0x1      0x0     0x0     2
    0x3      0x0     0x3     3
Ref. 0x20000, etiqueta=0x8,
              c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x3
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x0     5
    0x0      0x1     0x4     4
    0x0      0x2     0x10    6
    0x0      0x3     0x8     7
    0x1      0x0     0x0     2
    0x3      0x0     0x3     3
Ref. 0x80000, etiqueta=0x20,
              c=0x0, w=0x0
   ----Reemplazo fifo: conjunto:0x0,
                       linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x20    8
    0x0      0x1     0x4     4
    0x0      0x2     0x10    6
    0x0      0x3     0x8     7
    0x1      0x0     0x0     2
    0x3      0x0     0x3     3
 Estadisticas:
   Rerencias:                                  9
   Leas precargadas:                           0
   Exitos en referencias:                      2
   Exitos incluyendo lineas precargadas:       2
   Fallos incluyendo lineas precargadas:       7
   Fallos compulsorios total:                  6
```

3. Considerar una memoria principal de 64 Kb. estructurada en bytes y la siguiente secuencia de acceso a memoria principal:

   `04F5h, 11E0h, 1500h, 2000h, 241Fh, 16FFh, 1233h, 21F0h`

   Mostrar el contenido del directorio caché, así como los fallos y precargas que se producen para cada una de las cachés descritas en la siguiente tabla:

	Tamaño	Tam. línea	Organización
Caché 1	4 Kb.	256 bytes	Directa; Precarga siempre.
Caché 2	2 Kb.	256 bytes	Totalmente asociativa; Precarga bajo fallo; Reemplazo FIFO.
Caché 3	2 Kb.	256 bytes	Asoc. de 2 conjuntos; Sin precarga Reemplazo LRU.

```
############### CACHE 1:
Param.: n= 4, c= 4, w=8
Cache de mapeo directo.
Lineas en cache: 16
Conjuntos: 16
Lineas por conjunto: 1
Precarga:SIEMPRE,
Lineas a precargar:1

Ref. 0x4f5, etiqueta=0x0,
            c=0x4, w=0xf5
   ----Linea vacia: conjunto:0x4,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x4     0x0     0
Precarga: 0x5f5, etiqueta=0x0,
                 c=0x5, w=0xf5
   ----Linea vacia: conjunto:0x5,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x4     0x0     0
    0x5     0x0     0
Ref. 0x11e0, etiqueta=0x1,
             c=0x1, w=0xe0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x0
   ----Directorio:
    Lin.  Etq.    Ref.
   0x1     0x1     1
   0x4     0x0     0
   0x5     0x0     0
Precarga: 0x12e0, etiqueta=0x1,
                  c=0x2, w=0xe0
   ----Linea vacia: conjunto:0x2,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x1     0x1     1
    0x2     0x1     0
    0x4     0x0     0
```

```
    0x5     0x0     0
Ref. 0x1500, etiqueta=0x1,
             c=0x5, w=0x0
   ----Reemplazo:  linea:0x5
   ----Directorio:
    Lin.    Etq.    Ref.
    0x1     0x1     1
    0x2     0x1     0
    0x4     0x0     0
    0x5     0x1     2
Precarga: 0x1600, etiqueta=0x1,
                  c=0x6, w=0x0
   ----Linea vacia: conjunto:0x6,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x1     0x1     1
    0x2     0x1     0
    0x4     0x0     0
    0x5     0x1     2
    0x6     0x1     0
Ref. 0x2000, etiqueta=0x2,
             c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x1     1
    0x2     0x1     0
    0x4     0x0     0
    0x5     0x1     2
    0x6     0x1     0
Precarga: 0x2100, etiqueta=0x2,
                  c=0x1, w=0x0
   ----Reemplazo:  linea:0x1
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     0
    0x4     0x0     0
    0x5     0x1     2
    0x6     0x1     0
Ref. 0x241f, etiqueta=0x2,
             c=0x4, w=0x1f
   ----Reemplazo:  linea:0x4
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     0
    0x4     0x2     4
    0x5     0x1     2
    0x6     0x1     0
Precarga: 0x251f, etiqueta=0x2,
                  c=0x5, w=0x1f
   ----Reemplazo:  linea:0x5
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     0
Ref. 0x16ff, etiqueta=0x1,
             c=0x6, w=0xff
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
Precarga: 0x17ff, etiqueta=0x1,
                  c=0x7, w=0xff
   ----Linea vacia: conjunto:0x7,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
    0x7     0x1     0
Ref. 0x1233, etiqueta=0x1,
             c=0x2, w=0x33
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     6
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
    0x7     0x1     0
Precarga: 0x1333, etiqueta=0x1,
                  c=0x3, w=0x33
   ----Linea vacia: conjunto:0x3,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     1
    0x2     0x1     6
    0x3     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
    0x7     0x1     0
Ref. 0x21f0, etiqueta=0x2,
             c=0x1, w=0xf0
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     7
    0x2     0x1     6
    0x3     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
    0x7     0x1     0
Precarga: 0x22f0, etiqueta=0x2,
                  c=0x2, w=0xf0
   ----Reemplazo:  linea:0x2
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x2     3
    0x1     0x2     7
    0x2     0x2     6
    0x3     0x1     0
    0x4     0x2     4
    0x5     0x2     2
    0x6     0x1     5
    0x7     0x1     0

Estadisticas:
Referencias:                          8
Lineas precargadas:                   8
```

```
Exitos en referencias:                3
Exitos incluyendo lineas precargadas: 3
Fallos incluyendo lineas precargadas: 13
Fallos compulsorios total:            8


############## CACHE 2:
Param.: n= 3, c= 0, w=8
Lineas en cache: 8
Conjuntos: 1
Lineas por conjunto: 8
Reemplazo:FIFO
Precarga:BAJO FALLO
Lineas a precargar:1

Ref. 0x4f5, etiqueta=0x4, w=0xf5
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.   Ref.
    0x0     0x4    0
Precarga: 0x5f5, etiqueta=0x5,  w=0xf5
   ----Linea vacia: conjunto:0x0,
                 linea en conj.:0x1
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
Ref. 0x11e0, etiqueta=0x11,  w=0xe0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x2
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
Precarga: 0x12e0, etiqueta=0x12,  w=0xe0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x3
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
Ref. 0x1500, etiqueta=0x15,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x4
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
Precarga: 0x1600, etiqueta=0x16,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x5
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    0
Ref. 0x2000, etiqueta=0x20,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x6
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    0
    0x6     0x20    3
Precarga: 0x2100, etiqueta=0x21,  w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x7
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x4     0
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    0
    0x6     0x20    3
    0x7     0x21    0
Ref. 0x241f, etiqueta=0x24,  w=0x1f
   ----Reemplazo fifo: conjunto:0x0,
                       linea en conj.:0x0
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x24    4
    0x1     0x5     0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    0
    0x6     0x20    3
    0x7     0x21    0
Precarga: 0x251f, etiqueta=0x25,  w=0x1f
   ----Reemplazo fifo: conjunto:0x0,
                       linea en conj.:0x1
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x24    4
    0x1     0x25    0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    0
    0x6     0x20    3
    0x7     0x21    0
Ref. 0x16ff, etiqueta=0x16,  w=0xff
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x24    4
    0x1     0x25    0
    0x2     0x11    1
    0x3     0x12    0
    0x4     0x15    2
    0x5     0x16    5
    0x6     0x20    3
    0x7     0x21    0
Ref. 0x1233, etiqueta=0x12,  w=0x33
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x24     4
    0x1     0x25     0
    0x2     0x11     1
    0x3     0x12     6
    0x4     0x15     2
    0x5     0x16     5
    0x6     0x20     3
    0x7     0x21     0
Ref. 0x21f0, etiqueta=0x21,  w=0xf0
```

```
   ----Exito
   ----Directorio:
    Lin.    Etq.    Ref.
    0x0     0x24     4
    0x1     0x25     0
    0x2     0x11     1
    0x3     0x12     6
    0x4     0x15     2
    0x5     0x16     5
    0x6     0x20     3
    0x7     0x21     7
Estadisticas:
Referencias:                                8
Lineas precargadas:                         5
Exitos en referencias:                      3
Exitos incluyendo lineas precargadas:       3
Fallos incluyendo lineas precargadas:       10
Fallos compulsorios total:                  8
```

```
############## CACHE 3:
Param.: n= 3, c= 1, w=8
Lineas en cache: 8
Palabras por linea: 256
Conjuntos: 2
Lineas por conjunto: 4
Reemplazo:LRU
No precarga

Ref. 0x4f5, etiqueta=0x2, c=0x0, w=0xf5
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x2     0
Ref. 0x11e0, etiqueta=0x8,
             c=0x1, w=0xe0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x2     0
    0x1      0x0     0x8     1
Ref. 0x1500, etiqueta=0xa,
             c=0x1, w=0x0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x1
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x2      0
    0x1      0x0      0x8      1
    0x1      0x1      0xa      2
Ref. 0x2000, etiqueta=0x10,
             c=0x0, w=0x0
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x1
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x2     0
    0x0      0x1     0x10    3
    0x1      0x0     0x8     1
    0x1      0x1     0xa     2
Ref. 0x241f, etiqueta=0x12,
             c=0x0, w=0x1f
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x2
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x2      0
    0x0      0x1      0x10     3
    0x0      0x2      0x12     4
    0x1      0x0      0x8      1
    0x1      0x1      0xa      2
Ref. 0x16ff, etiqueta=0xb,
             c=0x0, w=0xff
   ----Linea vacia: conjunto:0x0,
                    linea en conj.:0x3
   ----Directorio:
    Cto.     Lin.     Etq.    Ref.
    0x0      0x0      0x2     0
    0x0      0x1      0x10    3
    0x0      0x2      0x12    4
    0x0      0x3      0xb     5
    0x1      0x0      0x8     1
    0x1      0x1      0xa     2
Ref. 0x1233, etiqueta=0x9,
             c=0x0, w=0x33
   ----Reemplazo lru: conjunto:0x0,
                      linea en conj.:0x0
   ----Directorio:
    Cto.     Lin.    Etq.    Ref.
    0x0      0x0     0x9     6
    0x0      0x1     0x10    3
    0x0      0x2     0x12    4

0x0     0x3     0xb     5
    0x1      0x0      0x8      1
    0x1      0x1      0xa      2
Ref. 0x21f0, etiqueta=0x10,
             c=0x1, w=0xf0
   ----Linea vacia: conjunto:0x1,
                    linea en conj.:0x2
   ----Directorio:
    Cto.     Lin.     Etq.     Ref.
    0x0      0x0      0x9      6
    0x0      0x1      0x10     3
    0x0      0x2      0x12     4
    0x0      0x3      0xb      5
    0x1      0x0      0x8      1
    0x1      0x1      0xa      2
    0x1      0x2      0x10     7
Estadisticas:
  Referencias:                              8
  Lineas precargadas:                       0
  Exitos en referencias:                    0
  Exitos incluyendo lineas precargadas:     0
  Fallos incluyendo lineas precargadas:     8
  Fallos compulsorios total:                7
```

4. Disponemos de una memoria caché de 4 Kb. con tamaño de línea de 256 bytes, organización asociativa de 8 conjuntos, algoritmo de precarga bajo fallo y algoritmo de reemplazo LRU.

   Conectamos la caché entre un procesador de 8 bits y una memoria principal de 1 Mb. Se pide:

a) Diseñar la memoria principal de manera que se optimice la transferencia de información entre caché y memoria principal.
b) Suponiendo la caché inicialmente vacía, describir la evolución del directorio caché para la siguiente secuencia de petición de direcciones a memoria principal:
`319F0h, 31AF0h, 7013Ch, 77777h, 44037h,`
`778DEh, A5021h`
Indicar claramente el ciclo en el que entra cada dato en el directorio caché, así como su etiqueta identificativa dentro de éste.
c) Comparar el sistema anterior con respecto a criterios de rendimiento y coste frente a una caché organizada de forma directa y a otra totalmente asociativa.

5. Considerar una memoria asociativa organizada en palabras de 8 bits. Realizar algoritmos para implementar las operaciones de búsqueda que se muestran a continuación, indicando a cada paso los valores del registro comparando (C) y del registro máscara (M). Todos los algoritmos deben ser independientes del número de celdas de la memoria.
a) Los números que no sean iguales al 01101110.
b) Los números múltiplos de 4, bien menores a -38 ó bien comprendidos entre 0 y 82, considerando representación signo-magnitud.
c) Los números comprendidos entre 20 y 30.

SOLUCIÓN:

(a) Buscamos los números que difieran en el primer bit, los que difieran en el segundo, ... (se puede hacer de diversas maneras, con esta intentamos hacer busquedas disjuntas de patrones).

1. M=1000 0000; C=1110 1110; Búsqueda. (Patrón 1xxxxxxx)
2. M=1100 0000; C=0010 1110; Búsqueda. (Patrón 00xxxxxx)
3. M=1110 0000; C=0100 1110; Búsqueda. (Patrón 010xxxxx)
4. M=1111 0000; C=0111 1110; Búsqueda. (Patrón 0111xxxx)
5. M=1111 1000; C=0110 0110; Búsqueda. (Patrón 01100xxx)
6. M=1111 1100; C=0110 1010; Búsqueda. (Patrón 011010xx)
7. M=1111 1110; C=0110 1100; Búsqueda. (Patrón 0110110x)
8. M=1111 1111; C=0110 1111; Búsqueda. (Patrón 01101111)

(b) En binario: +82=0 1010010 y -38=1 0100110. Buscamos primero los positivos menores a 82 y multiplos de 4. Denotamos con 0 el bit menos significativo y con 7 el más significativo:

1. C=00000000, M=11000011; Búsqueda. (Patrónes 00xxxx00)
2. C=01000000, M=11110011; Búsqueda. (Patrónes 0100xx00)

3. C=01010000, M=11111111; Búsqueda. (Patrón 01010000)

Y luego los negativos: es decir aquéllos cuyos últimos 7 bits corresponda a un valor mayor a 38, su bit más significativo sea 1 y los bits 1 y 0 sean 0.

4. C=11000000, M=11000011; Búsqueda. (Patrón 11xxxx00)
5. C=10110000, M=11110011; Búsqueda. (Patrón 1011xx00)
6. C=10101000, M=11111011; Busqueda. (Patrón 10101x00)

6. Supongamos una caché totalmente asociativa compuesta por 4 líneas, y un programa compuesto por un bucle que se ejecuta 1000 veces y solicita las líneas A-B-C-D-E de memoria principal en cada iteración, siguiendo la secuencia A, B, C, D, E, A, C. Sabiendo que el número de accesos a cada línea de memoria principal por cada iteración del bucle es A:100, B:5, C:6, D:4 y E:10, calcular el índice de fallos si utilizamos el algoritmo de reemplazo LRU. Idem para el algoritmo FIFO.
7. Sea un procesador de 16 bits con un bus de direcciones de 24 bits al que se le quiere dotar de un sistema de memoria caché con las siguientes características: Organización totalmente asociativa, 8 líneas de 16 palabras cada una, precarga bajo fallo y algoritmo de reemplazo FIFO.
   a) Diseñar un sistema de memoria principal a partir de pastillas de 256Kx8 de forma que la transferencia de información entre memoria principal y caché sea lo más rápida posible.
   b) Dado el estado inicial de la caché que aparece en la siguiente tabla, indicar la evolución del directorio caché para la siguiente secuencia de referencias a memoria: 0A3014h, 12345Bh, BB2303h, C0E50Fh, C0E4F3h. Las líneas entraron en la caché siguiendo el orden que aparece en la tabla. Suponer que una línea precargada cuenta como una línea cualquiera a la hora de aplicar el algoritmo FIFO.

Número de línea	Etiqueta para los datos de esa línea
0	0A301h
1	43A02h
2	12345h
3	AC0C3h
4	C0E50h
5	BB232h
6	11E91h
7	11E92h

8. Considera el siguiente código ensamblador del i8086, en el que se facilita el desplazamiento (offset), referido a cada segmento, de las etiquetas y/o instrucciones:

```
0000              DATOS SEGMENT
0000                 TABLA1 DW 128 DUP(1)
0100                 TABLA2 DW 128 DUP(0)
0200              DATOS ENDS
                  ;-------------------------
0000              PILA SEGMENT STACK
0000                  DB 512 DUP(0)
0200              PILA ENDS
                  ;-------------------------
0000              CODIGO SEGMENT
                  ASSUME CS:CODIGO,DS:DATOS,SS:PILA
0000              INICIO: MOV AX, SEG DATOS
0003                      MOV DS, AX
0005                      XOR SI, SI
0007              BUCLE:  MOV BX, TABLA1[SI]
000B                      ADD BX, TABLA2[SI]
000F                      MOV TABLA1[SI], BX
0013                      ADD SI, 02H
0016                      CMP SI, 128
001A                      JNE BUCLE
001C              FINP:   MOV AH,4CH
001E                      INT 21H
0020              CODIGO ENDS
                         END INICIO
```

a) Escribe la traza de la ejecución mostrando las palabras de memoria referenciadas diciendo si se trata de búsqueda de instrucción, lectura o escritura de datos. Suponga para ello que los registros de segmentos son: DS=200H, SS=220H, CS=240H.

b) Dado que el bus de datos es de 16 bits, diseñe un esquema de memoria entrelazada para este procesador que conste de 2 bancos de memoria cada uno direccionado a nivel de byte.

c) Con una pequeña caché de 1Kbyte, con tamaño de palabra de 2 bytes, tamaño de línea de 4 palabras, ¿cuál es la tasa de fallos para la ejecución de este programa en los siguientes casos: completamente asociativa, de mapeo directo, asociativa por conjuntos con 8 conjuntos? Suponga que no hay precarga y la actualización es write-through.

d) ¿Y si dicha caché se dividiera en dos: una para referencias a datos y otra para instrucciones?

SOLUCIÓN:

Nos vamos a centrar en la ejecución del bucle. Puesto que éste se ejecuta 64 veces solo vamos a analizar las 2 primeras iteraciones: (Con R indicamos lectura de datos, W escritura de datos e I búsqueda de instrucción). Las direcciones corresponden a posiciones físicas del byte referenciado. (Para el segmento de código la posición física se calcula como CS*10H+offset y para el de datos DS*10H+offset). Se indica el

numero de bytes transferidos en cada referencia.

```
 1 iteración:  I 2407H  ; instrucción MOV BX, TABLA1[SI]   4 bytes
               R 2000H  ; comienzo TABLA1                  2 bytes
               I 240BH  ; instrucción MOV BX, TABLA2[SI]   4 bytes
               R 2100H  ; comienzo TABLA2                  2 bytes
               I 240FH  ; instrucción MOV TABLA1[SI], BX   4 bytes
               W 2000H  ; comienzo TABLA1                  2 bytes
               I 2413H  ; instrucción ADD SI, 02H          3 bytes
               I 2416H  ; instrucción CMP SI, 128          4 bytes
               I 241AH  ; instrucción JNE                  2 bytes
 2 iteracion:  I 2407H  ; instrucción MOV BX, TABLA1[SI]
               R 2002H  ; comienzo TABLA1 + 2
               I 240BH  ; instrucción MOV BX, TABLA2[SI]
               R 2102H  ; comienzo TABLA2 + 2
               I 240FH  ; instrucción MOV TABLA1[SI], BX
               W 2002H  ; comienzo TABLA1 + 2
               I 2413H  ; instrucción ADD SI, 02H
               I 2416H  ; instrucción CMP SI, 128
               I 241AH  ; instrucción JNE
            ......................................
```

A la hora de simular la traza anterior hay que tener en cuenta las siguientes indicaciones:

- Los parametros a fijar: n=9; w=2;

- Obtener la direccion de palabra asociada a cada referencia ya que las peticiones están expresadas en direcciones a nivel de byte.

- Como las palabras son de 2 bytes pero las referencias son de bytes, el acceso a una palabra que empieza en una posicion par solo implica un acceso a memoria, mientras que el acceso a una palabra que empieza en una posicion impar necesita dos accesos a memoria.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Apéndice E

Solución a las relaciones de problemas

E.1. REPRESENTACIÓN DE LA INFORMACIÓN

Representación de los datos

1. Convierte los siguientes números decimales a binario: 1984, 4000, 8192.
   Solución: 11111000000, 111110100000,
   2^{13}=10000000000000
2. ¿Qué número es $1001101001_{(2}$ en decimal? ¿En octal? ¿En hexadecimal?
   Solución: $1001101001_{(2} = 617_{(10} = 1151_{(8} = 269_{(16}$
3. ¿Cuáles de éstos son números hexadecimales válidos?: CAE, ABAD, CEDE, DECADA, ACCEDE, GAFE, EDAD. Pasar los números válidos a binario.
   Solución:
   - CAE = 1100 1010 1110
   - ABAD = 1010 1011 1010 1101
   - CEDE = 1100 1110 1101 1110
   - DECADA = 1101 1110 1100 1010 1101 1010
   - ACCEDE = 1010 1100 1100 1110 1101 1110
   - EDAD = 1110 1101 1010 1101
4. Expresa el número decimal 100 en todas las bases, de 2 a 9.
   Solución: $100_{(10} = 121_{(9} = 144_{(8} = 202_{(7} = 244_{(6} = 400_{(5} = 1210_{(4} = 10201_{(3} = 1100100_{(2}$
5. ¿Cuántos enteros positivos se pueden representar con k dígitos usando

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

números en base r?

Solución: Variaciones con repetición de r elementos tomados de k en k, es decir, $RV_r^k = r^k$.

6. El siguiente conjunto de 16 bits:

   1001 1000 0101 0100

   puede representar un número que depende del convenio utilizado. Dar su significación decimal en el caso de que el convenio sea:

   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2
   d) BCD natural (8,4,2,1)
   e) BCD exceso a 3
   f) BCD Aiken (2,4,2,1)

   Solución:

   a) Signo y Magnitud = -6228
   b) Complemento a 1 = -26539
   c) Complemento a 2 = -26540
   d) BCD natural (8,4,2,1) = 9854
   e) BCD exceso a 3 = 6521
   f) BCD Aiken (2,4,2,1) = sólo válido el 4 final

7. Para las siguientes representaciones de n bits, dar el rango, precisión, representaciones del 0 y comparar el número de números positivos con el de negativos.

   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2

   Solución:

Formato	Rango	Precisión	Cero(s)
Sig/Mag	$[-(2^{n-1}-1), 2^{n-1}-1]$	$2^n - 1$	10..0/00..0
C1	$[-(2^{n-1}-1), 2^{n-1}-1]$	$2^n - 1$	11..1/00..0
C2	$[-2^{n-1}, 2^{n-1}-1]$	2^n	00..0

8. Expresar los números decimales -63, 91 y -23 en signo/magnitud, en complemento a uno y en complemento a dos utilizando una palabra de 8 bits.

   Solución:

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Formato	N.º Positivos	N.º Negativos
Sig/Mag	$2^{n-1}-1$	$2^{n-1}-1$
C1	$2^{n-1}-1$	$2^{n-1}-1$
C2	$2^{n-1}-1$	2^{n-1}

Número	Sig/Magnitud	C1	C2
-63	1011 1111	1100 0000	1100 0001
91	0101 1011	0101 1011	0101 1011
-23	1001 0111	1110 1000	1110 1001

9. ¿Cuántos bits son necesarios para representar todos los números entre -1 y +1 con un error no mayor de $0.0001_{(10}$ en complemento a dos y con notación en punto fijo?

   Solución: $n = ceil\left(\log_2(\frac{2}{10^{-4}}+1)\right) = 15$

10. Representar los siguientes números como flotantes IEEE-754 en simple precisión.

    a) 10
    b) 10.5
    c) 0.1
    d) 0.5
    e) -23.15625

    Solución:

    a) 10 = | 0 | 1000 0010 | 0100 0000 ... 0 |
    b) 10.5 = | 0 | 1000 0010 | 0101 0000 ... 0 |
    c) 0.1 = | 0 | 0111 1011 | 1001 1001 1001 1001 ... 1001 |
    d) 0.5 = | 0 | 0111 1110 | 0000 ... 0 |
    e) -23.15625 = | 1 | 1000 0011 | 0111 0010 1000 0000 ... 0 |

11. ¿Qué inconveniente encontramos en la representación de números reales en punto fijo? Dados estos dos números en el formato IEEE 754: C0E80000 y 00080000; decir qué valores representan en decimal.

    Solución:

    Limita el rango y la precisión y presenta el problema del escalado.

    $C0E80000_{(H}$ = -7.25$_{(10}$; $00080000_{(H}$ = 1 x $2^{-130}_{(10}$

12. ¿Cuáles son los requisitos deseables de un sistema de numeración de enteros con signo? ¿Qué sistema cumple mejor esos requisitos? Expresar el número 9.75 en su representación IEEE 754 de 32 bits (convierte a

hexadecimal la palabra IEEE de 32 bits).

Solución:

Requisitos: igual número de valores positivos y negativos, detección simple del signo y del cero e implementación sencilla de las operaciones aritméticas básicas. El Complento a 2.

0	1000 0010	0011 1000 0000 ... 0	$= 411C0000_{(H}$

13. En una PDP-11 los números en punto flotante de precisión sencilla tienen un bit de signo, un exponente en exceso a 128 y una mantisa de 24 bits. Se exponencia a la base 2. El punto decimal está en el extremo izquierdo de la fracción. Debido a que los números normalizados siempre tienen el bit de la izquierda a 1, éste no se guarda; solo se guardan los 23 bits restantes. Expresa el número 7/64 en este formato.

    Solución:

0	0111 1100	1100 0000 ... 0

14. Expresa el número 7/64 en el formato IBM/360, y en el formato IEEE-754.

    Solución:

a) IBM/360 :

0	100 0000	0001 1100 0000 ... 0

b) IEEE-754 :

0	0111 1011	1100 0000 0000 ... 0

15. Expresa el número en punto flotante de 32 bits $3FE00000_{(16}$ como número decimal si está representado en los siguientes formatos de 32 bits:

    a) IEEE-754
    b) IBM/360
    c) PDP-11

    Solución:

    a) IEEE-754 : 1.75
    b) IBM/360 : 0.0546875
    c) PDP-11 : 0.875

16. Los siguiente números en punto flotante constan de un bit de signo, un exponente en exceso a 64 y una mantisa de 16 bits. Normalízalos suponiendo que la exponenciación es a la base 2 y que el formato NO es del tipo 1.XXXX... con el "1" implíto.

    - 0 1000000 0001010100000001
    - 0 0111111 0000001111111111
    - 0 1000011 1000000000000000

    Solución:

    - 0 0111101 1010100000001000
    - 0 0111001 1111111111000000

- Ya está *normalizado*

17. ¿Cuál son los números positivos más pequeño y más grande representables para los siguientes convenios?:
    a) IEEE-754
    b) IBM/360
    c) PDP-11

    Solución:

    a) IEEE-754 : m = 1.0 x 2^{-126} (Mirar nota al pie [1]); M $\approx$ 2 x 2^{127}
    b) IBM/360 : m = 0.0001 x 16^{-64}; M $\approx$ 1 x 16^{63}
    c) PDP-11 : m = 1.0 x 2^{-128}; M $\approx$ 2 x 2^{127}

Representación de las instrucciones

1. Diseña un código de operación con extensión que permita lo siguiente y se pueda codificar en una instrucción de 36 bits:
    a) 7 instrucciones con dos direcciones de 15 bits y un número de registro de 3 bits,
    b) 500 instrucciones con una dirección de 15 bits y un número de registro de 3 bits,
    c) 50 instrucciones sin direcciones ni registros.

    Solución:

    a) 7 instrucciones con dos direcciones de 15 bits y un número de registro de 3 bits:

C.O. (3)	*Dirección1 (15)*	*Dirección2 (15)*	*Registro (3)*

    con C.O. de 001 (1) a 111 (7).

    b) 500 instrucciones con una dirección de 15 bits y un número de registro de 3 bits:

000 (3)	*C.O. (9)*	*Dirección 1 (15)*	*Registro (3)*	*XXX XXX (6)*

    con C.O. de 000000001 (1) a 111110100 (500).

    c) 50 instrucciones sin direcciones ni registros:

000 (3)	*000000000 (9)*	*C.O.(6)*	*XXX ... X (18)*

    con C.O. de 000000 (0) a 110001 (49).

2. ¿Es posible diseñar un código de operación con extensión que permita codificar lo siguiente en una instrucción de 12 bits? Un registro se direcciona con 3 bits.
    a) 4 instrucciones con tres registros,

[1] Si consideramos números desnormalizados m= 1.0 x 2^{-149}

b) 255 instrucciones con un registro,
c) 16 instrucciones con cero registros.

Solución: No. Si hacemos el diseño vemos que nos falta un bit.

a) 4 instrucciones con tres registros:

C.O. (3)	*R1 (3)*	*R2 (3)*	*R3 (3)*

con C.O. de 000 a 011.

b) 255 instrucciones con un registro:

1 (1)	*C.O. (8)*	*REG (3)*

con C.O. de 00000000 a 11111110.

c) 16 instrucciones con cero registros:

1 (1)	*11111111 (8)*	*C.O. (3)*

Sólo nos quedan 3 bits, con lo que se podrán codificar como máximo 8 instrucciones con cero registros.

3. Cierta máquina tiene instrucciones de 16 bits y direcciones de 6. Algunas instrucciones tienen una dirección y otras dos. Si hay n instrucciones de dos direcciones, ¿cuál es el número máximo de instrucciones de una dirección?

   Solución: $(16 - n) \cdot 64$

4. Queremos diseñar el formato de instrucciones de un microprocesador con 16 registros internos, y que puede direccionar 1 Kbyte de memoria. Dentro del conjunto de instrucciones encontramos las siguientes:

   a) 15 instrucciones del tipo:

Código de Op.	Desp, Dirección, Dirección

   b) 63 instrucciones del tipo:

Código de Op.	Desp, Dirección, Registro

   c) 60 instrucciones del tipo:

Código de Op.	Dirección, Registro

   d) 15 instrucciones del tipo:

Código de Op.	Registro, Registro, Registro

   e) 16 instrucciones del tipo:

Código de Op.

   Donde el campo desplazamiento (*Desp*) nos permite implementar el direccionamiento relativo, desplazándonos como máximo 63 bytes respecto de la dirección base apuntada por la primera *Dirección*. El desplazamiento siempre será positivo.

   Se pide:

a) Decir de cuántas direcciones es cada tipo de instrucciones (0, 1, 2 ó 3).

b) Diseñar el formato de cada tipo de instrucciones mediante códigos de operación con extensión.

Solución:

- *Decir de cuántas direcciones es cada tipo de instrucciones (0, 1, 2 ó 3).*
  (a) 2, (b) 2, (c) 2, (d) 3, (e) 0.
- *Diseñar el formato de cada tipo de instrucciones mediante códigos de operación con extensión.*
  Desplazamiento → 6 bits, Dirección → 10 bits, Registro → 4 bits.

a)

C.O. (4)	Desp (6)	Dir (10)	Dir (10)

b)

1111 (4)	C.O. (6)	Desp (6)	Dir (10)	Reg (4)

c)

1111 (4)	111111 (6)	C.O. (6)	Dir (10)	Reg (4)

con C.O. de 000000 (0) a 111011 (59).

d)

1111 (4)	111111 (6)	1111XX (6)	C.O.(2)	Reg(4)	Reg(4)	Reg(4)

Para codificar las 16 instrucciones usamos XX de 00 a 11 (correspondientes a los códigos de operación 60 a 63 del formato anterior) junto con los dos bits indicados. Reservamos XX=11 y CO=11 para especificar el siguiente formato:

e)

1111(4)	111111(6)	111111(6)	11 (2)	C.O.(4)	XX...X(8)

Se necesitarán por lo tanto 30 bits para codificar todas las posibles instrucciones.

5. Dados los contenidos de las celdas de memoria que siguen y una máquina de una dirección con un acumulador, ¿qué valores cargan en el acumulador las instrucciones siguientes?
   - la palabra 20 contiene 40
   - la palabra 30 contiene 50
   - la palabra 40 contiene 60
   - la palabra 50 contiene 70

   a) LOAD 20
   b) LOAD [20]
   c) LOAD [[20]]
   d) LOAD 30
   e) LOAD [30]
   f) LOAD [[30]]

   Solución:

a) LOAD 20 $\Rightarrow AC \leftarrow 20$
b) LOAD [20] $\Rightarrow AC \leftarrow 40$
c) LOAD [[20]] $\Rightarrow AC \leftarrow 60$
d) LOAD 30 $\Rightarrow AC \leftarrow 30$
e) LOAD [30] $\Rightarrow AC \leftarrow 50$
f) LOAD [[30]] $\Rightarrow AC \leftarrow 70$

6. Compara las máquinas de 0, 1, 2, y 3 direcciones escribiendo programas para calcular

$$X = (A + B \times C)/(D - E \times F)$$

para cada una de las cuatro máquinas. Para que no se pierdan los valores originales de las variables A, B, C, D, E y F podremos apoyarnos en una variable temporal T. Suponiendo direcciones de 16 bits, códigos de operación de 8 bits y longitudes de instrucción que son múltiplos de 4 bits, ¿Cuántos bits necesita cada computadora para calcular X?

Solución:

- *3 Direcciones*

```
MUL B, C, T
ADD A, T, T
MUL E, F, X
SUB D, X, X
DIV T, X, X
```

Bits necesarios: 5 x (8 + 3 x 16) = 280.

- *2 Direcciones*

```
MOVE C, T
MUL B, T
ADD A, T
MOVE F, X
MUL E, X
SUB D, X
DIV T, X
```

Bits necesarios: 7 x (8 + 2 x 16) = 280.

- *1 Dirección*

```
LOAD C
MUL B
ADD A
STORE T
LOAD F
```

```
MUL E
SUB D
DIV T
STORE X
```

Bits necesarios: 9 x (8 + 16) = 216.

- *0 Direcciones (usando una pila)*

```
PUSH A
PUSH B
PUSH C
MUL
ADD
PUSH D
PUSH E
PUSH F
MUL
SUB
DIV
POP X
```

Bits necesarios: 5 x 8 + 7 x (8 + 16) = 208.

E.2. PROCESADOR CENTRAL

1. Para el número de 16 bits: 1001 0101 1100 0001 almacenado en el registro AX, muestra el efecto de:
   a) `SHR AX, 1`
   b) `SAR AX, 1`
   c) `SAL AX, 1`
   d) `ROL AX, 1`
   e) `ROR AX, 1`

   Solución:

   a) `SHR AX, 1` ⇒ 0100 1010 1110 0000 CF=1
   b) `SAR AX, 1` ⇒ 1100 1010 1110 0000 CF=1
   c) `SAL AX, 1` ⇒ 0010 1011 1000 0010 CF=1
   d) `ROL AX, 1` ⇒ 0010 1011 1000 0011 CF=1
   e) `ROR AX, 1` ⇒ 1100 1010 1110 0000 CF=1
2. ¿Cómo podrías poner a 0 el registro AX si no dispones de una instrucción de BORRAR (CLEAR)?

   Solución: De menos a más eficientemente:

a) `SHL AX, 16`
b) `MOV AX, 0`
c) `SUB AX, AX`
d) `XOR AX, AX`

3. Inventa un método para intercambiar dos registros (AX y BX) sin usar un tercer registro o variable ni la instrucción XCHG. *Sugerencia:* piensa en la instrucción OR EXCLUSIVO.

   Solución:

```
XOR AX, BX
XOR BX, AX
XOR AX, BX
```

4. Tenemos 250 números en C2 de 16 bits en las posiciones 500 a 998 de memoria. Escribir un programa en ensamblador del 8086 que cuente los números $P \geq 0$ y $N < 0$ y que almacene P en la posición 1000 de memoria y N en la 1002.

   Solución:

```
        XOR AX, AX ; P=0
        XOR BX, BX ; N=0
        MOV DI, 500
        MOV CX, 250
LAZO:   MOV DX, [DI]
        ADD DI, 2
        CMP DX, 0
        JL NEGA ; Salta si menor que cero
        INC AX
        JMP CONT
NEGA:   INC BX
CONT:   LOOP LAZO
        MOV [1000], AX
        MOV [1002], BX
        END
```

Otro posible algoritmo sólo contaría los positivos y luego calcularía los negativos restando de 250. Esto es más eficiente ya que nos evitamos los incrementos del registro `BX`, un salto incondicional y el `XOR BX, BX`. También podemos hacer la comparación con el dato de memoria directamente sin copiarlo en un registro, ya que la instrucción `CMP` no modifica los operandos (sólo actualiza los flags despues de restar, pero no guarda el resultado en ningún sitio):

```
        XOR AX, AX ; P=0
        MOV DI, 500
        MOV CX, 250
LAZO:   CMP [DI], 0
        JL CONT
        INC AX
CONT:   ADD DI, 2
        LOOP LAZO
        MOV [1000], AX
        SUB AX, 250; AX=AX-250
        NEG AX
        MOV [1002], AX
        END
```

5. Tenemos 100 números positivos de 16 bits en las posiciones de memoria 1000 a 1198. Escribir un programa que determine cuál es el número mayor de todos y lo almacene en la posición 1200.

   Solución:

```
        MOV CX, 99
        MOV DI, 1000
        MOV AX, [DI]; tomar el primer numero
        ADD DI, 2
LAZO:   CMP AX, [DI]
        JG SIGUE
        MOV AX, [DI]
SIGUE:  ADD DI, 2
        LOOP LAZO
        MOV [DI], AX
        END
```

6. Escribir un procedimiento en lenguaje ensamblador del 8086 que calcule N! y lo almacene en la posición de memoria 502. Supondremos que 1<N<255, que el número N está en la posición 500 de memoria y que N! cabe en un registro de 8 bits.

   Solución:

```
        XOR CX, CX
        MOV AL, [500]
        MOV CL, AL
        DEC CL
BUCLE:  MUL CL
        LOOP BUCLE
        MOV [502], AX
        END
```

7. Escribir una subrutina en lenguaje ensamblador que convierta un entero binario positivo (menor que 1000 y almacenado en la posición 2000 de memoria) a ASCII, guardando cada carácter a partir de la posición 3000 de memoria. Por ejemplo, el número 345 debe traducirse a los caracteres ASCII 3, 4 y 5 (33H, 34H y 35H en hexadecimal respectivamente)

   Solución:

```
        MOV DI, 3002
        MOV CX, 3 ; cifras
        MOV AX, [2000]
        MOV BX, 10
BUCLE:  DIV BL
        ADD AH, 30h
        MOV [DI], AH
        DEC DI
        MOV AH, 0
        LOOP BUCLE
        END
```

8. a) Escribe el diagrama de flujo y el programa en ensamblador del 8086 de Intel para el algoritmo que realiza la siguiente función:

   Dada una tabla en la posición 100 de memoria con 50 números de 16 bits en complemento a dos, multiplicar por 2 los números pares y dividir por 2 los impares (división entera), dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar las instrucciones MUL y DIV. No considerar el caso de *overflow* al multiplicar por dos.

   b) ¿Qué diferencia hay entre los desplazamientos lógicos y aritméticos a la derecha?

   Solución:

   a) El diagrama de flujo y el código resultante pueden verse a continuación. Hay que tener en cuenta que en los números impares negativos se pierde el bit menos significativo (a 1), con lo que el resultado quedará en C1. Habrá que sumar uno en este caso para tenerlo en C2.

```
        MOV SI, 100
        MOV CX, 50
LAZO:   MOV AX, [SI]
        SAR AX, 1
        JC IMPAR
PAR:    SAL AX, 2
        JMP CONT
IMPAR:  CMP AX, 0
        JGE CONT ; salta si es positivo
        ADD AX, 1
```

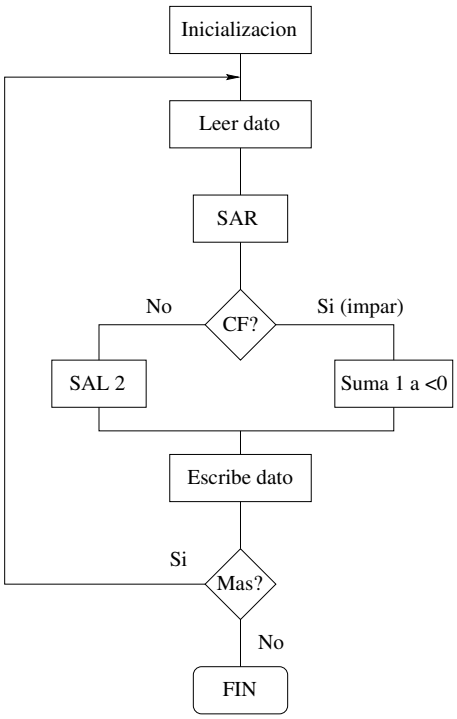


Figura E.1: Diagrama de Flujo del programa del problema 8

```
CONT:  MOV [SI], AX
       ADD SI, 2
       LOOP LAZO
       END
```

b) En los desplazamientos aritméticos a la derecha se produce extensión de signo y en los lógicos se introducen 0s. Mirar ejercicio 1.

9. Escribe el diagrama de flujo y el programa en ensamblador del 8086 de Intel para el algoritmo que realiza la siguiente función:
   Dada una tabla en la posición 500 de memoria con 50 números de 16 bits en complemento a dos, forzar a cero los números negativos y multiplicar por tres los positivos, dejando cada número modificado en la misma posición de la tabla en la que estaba. No utilizar ninguna instrucción de multiplicar (MUL). No considerar el caso de *overflow* al multiplicar por tres.
   Solución:

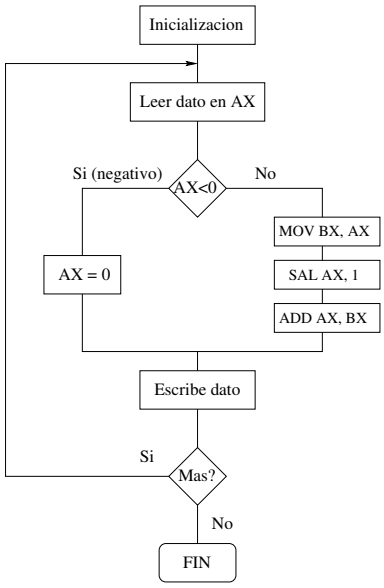


Figura E.2: Diagrama de Flujo del programa del problema 9

```
        MOV SI, 500
        MOV CX, 50
LAZO:   MOV AX, [SI]
        CMP AX, 0
        JG POSI
        MOV AX, 0
        JMP CONT
POSI:   MOV BX, AX
        SAL AX, 1
        ADD AX, BX
CONT:   MOV [SI], AX
        ADD SI, 2
        LOOP LAZO
        END
```

10. Suponga que el contenido de la memoria de un sistema basado en el i8086 es el que se muestra en la tabla E.1. El valor de los registros es: DS=SS=E000H, SI=0001H, DI=0002H, BP=0003H, BX=0004H. Se define así mismo una etiqueta `TABLA`, que apunta a la dirección física E0001H y que pertence al segmento de datos.

    Para cada una de las siguientes instrucciones, determine como quedan afectados los registros correspondientes:

posición física	contenido	
.....	.....	
E0006	DE H	
E0005	BC H	
E0004	9A H	
E0003	78 H	
E0002	56 H	
E0001	34 H	<-- TABLA
E0000	12 H	
.....	.....	
5D276	07 H	
5D275	06 H	
5D274	05 H	
5D273	04 H	
5D272	03 H	
5D271	02 H	
5D270	01 H	
.....	.....	

Tabla E.1: Contenido de la memoria

```
MOV AX, [TABLA]
MOV AX, [SI]
MOV AL, [SI]
MOV AX, [SI+2]
MOV AX, TABLA[SI]
MOV AL, TABLA[SI]
MOV AH, TABLA[SI]
MOV AX, [BP+2]
MOV AX, [BX+SI]
MOV AX, [BP+SI]
MOV AL, [BP+SI+1]
MOV AH, TABLA[BX][SI]
LEA AX, TABLA
LEA DI, TABLA[SI+1]
MOV AX, SEG TABLA
MOV CX, OFFSET TABLA
```

Solución:

```
MOV AX, [TABLA] ; AX = 5634H
MOV AX, [SI] ; AX = 5634H
MOV AL, [SI] ; AL = 34H
MOV AX, [SI+2] ; AX = 9A78H
MOV AX, TABLA[SI] ; AX = 7856H
MOV AL, TABLA[SI] ; AL = 56H
```

```
MOV AH, TABLA[SI] ; AH = 56H
MOV AX, [BP+2] ; AX = DEBCH
MOV AX, [BX+SI] ; AX = DEBCH
MOV AX, [BP+SI] ; AX = BC9AH
MOV AL, [BP+SI+1] ; AL = BCH
MOV AH, TABLA[BX][SI]  ; AH = DEH
LEA AX, TABLA ; AX = 0001H
LEA DI, TABLA[SI+1] ; DI = 0003H
MOV AX, SEG TABLA ; AX = E000H
MOV CX, OFFSET TABLA ; CX = 0001H
```

E.3. SECCIÓN DE CONTROL

Nos limitamos en este apartado a presentar los contenidos de la ROM de microprograma y la ROM de Proyección para cada uno de los ejercicios. Los bits no especificados se suponen puestos a cero.

ROM de Proyección del problema 4

Dir	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Cont.				8	10			4		5						

ROM de Proyección del problema 6

Dir	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Cont.										11		7		5		3

ROM de Proyección del problema 7

Dir	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Cont.	6	11		4		9										

Nota: En las siguientes tablas en la columna “Descripción” se muestra en pseudocódigo la acción que realiza cada microinstrucción. La abreviatura DEC. se ha empleado para indicar la decodificación, que implica la carga del contador de microprograma con el contenido de la posición de la ROM de proyección seleccionada por el registro de instrucción (RI), es decir, la operación $\mu PC \Leftarrow ROM_{\text{proyección}}(RI)$.

		L		R			C	De		X			Mem			ALU		F	M	Salto	
Comentario	Dir.	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	CS	Dir. Salto
(PC) → RI	0												0	1	1						
PC++; DEC.	1			0	1	1	1			1	0	1							1	1 1	
Z?	2																			1 0	00000 00100
PC++. Salto Fetch	3			0	1	1	1			1	0	1								1 1	00000 00000
(PC) → D	4												1	0	0						
D → PC	5			0	1	0				1	0	1								1 1	00000 00000
A-B. C?	6	1	0	0	0	1	1									0	1	1		0 1	00000 01000
Salto Fetch	7																			1 1	00000 00000
A → B. Salto	8	1	0							0	1	1								1 1	00000 00000
A→ I	9	1	0							0	0	1									
B→ A	10	1	1							0	1	0									
I → B	11	0	1							0	1	1								1 1	00000 00000

Tabla E.2: Memoria de microprograma para los problemas 1, 2 y 3

		Registros								ALU				PC			Mem.		MAR	DR			Salto	
Comentario	**Dir.**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**	**c10**	**c11**	**c12**	**c13**	**c14**	**c15**	**c16**	**c17**	**c18**	**c19**	**c20**	**CS**	**Dir. Salto**
PC → MAR	**0**													**0**	**0**	**1**			**1**				
PC++, MEM → DR	**1**													**1**	**0**	**0**	**1**	**1**					
DR → RI	**2**	**0**	**0**	**0**	**1**															**1**			
DEC.	**3**									**1**												**11**	**0**
A → TMP	**4**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**		**1**	**0**	**0**									**11**	**0**
A → MAR	**5**					**1**	**0**	**0**	**1**		**1**	**0**	**0**						**1**				
MEM → DR	**6**																**1**	**1**					
DR → B	**7**	**0**	**1**	**0**	**0**															**1**		**11**	**0**
Z?	**8**																					**10**	**0**
A → PC	**9**					**1**	**0**	**0**	**1**		**1**	**0**	**0**	**0**	**1**	**0**						**11**	**0**
A → MAR	**10**					**1**	**0**	**0**	**1**		**1**	**0**	**0**						**1**				
MEM → DR	**11**																**1**	**1**					
DR → TMP	**12**	**0**	**0**	**1**	**0**															**1**			
B× TMP → B	**13**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**		**1**	**1**	**0**									**11**	**0**

Tabla E.3: Problema 4. Memoria de microprograma

		LOAD			CLR	BUS		INC	SH	M	CS	DIR	
Comentario	**Dir.**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**	**CS**	**Dir. Salto**	
Clear AC y Cont.	**100**				**1**								
Si $\overline{x(0)}$ salta a 103	**101**										**10**	**0110 0111 (103)**	
Suma	**102**		**1**										
Shift, Inc, FIN?	**103**							**1**	**1**		**01**	**0000 0000**	
Salta 101	**104**										**11**	**0110 0101 (101)**	

Tabla E.4: Problema 5. Memoria de microprograma

		Banco de Reg.					AC, RI, DR		ALU			SR	Mem.		Cte			Salto	
Comentario	**Dir.**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**	**c10**	**c11**	**c12**	**c13**	**c14**	**c15**	**c16**	**c17**	**c18**	**c19**	**CS**	**Dir. /Cte.**
DR ⇐ (PC); AC ⇐ PC	**0**	1	1	1	1	1		1									1	1				
RI ⇐ DR	**1**								1		1											
PC ⇐ AC+0+c13; DEC.	**2**	1	1				1							1				1	1	1	1	
AC ⇐ A	**3**					1		1														
B ⇐ B+AC	**4**	0	1			1	1									1					1	1	0
AC ⇐ A	**5**					1		1														
B ⇐ AC + 0 (cte)	**6**	0	1				1												1		1	1	0
AC ⇐ A	**7**					1		1														
B ⇐ B+AC	**8**	0	1			1	1															
DR ⇐ (B); AC⇐ 0	**9**			0	1			1									1	1	1			0
B ⇐ DR+AC	**10**	0	1				1			1											1	1	0
Si N=1 salta a 0	**11**																			0	1	0
AC⇐ C; Si Z=1 salta a 0	**12**	1	0			1		1												1	0	0
PC⇐ AC+0	**13**	1	1				1												1		1	0

Tabla E.5: Problema 6. Memoria de microprograma

Descripción	Dir.	Mem.		Reg. Dir.			Reg. Dat.			Carga				ALU	Transf.			M	Salto	
		c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c18	CS	Dir. Salto
RI ⇐ (PC)	0	1	0											1							
TEMPA ⇐ PC	1										1					1					
TEMPB ⇐ R1	2							1	1			1					1				
PC⇐ALU; DEC.	3			0	0	1												1	1	11	
TMP ⇐ [RY]	4			1	1	1	1										1				
[RX] ⇐ TMP	5			1	1					1						1				11	0
TEMPA ⇐ [RX]	6										1						1				
TEMPB ⇐ [RY]	7						1					1					1				
[RX] ⇐ ALU	8									1			1		1			1		11	0
Si C=1 salta a 0	9																			01	0
PC ⇐ [RY]	10					1	1										1			11	0
TEMPA ⇐ SI	11				1						1					1					
TEMPB ⇐ [RY]	12						1					1					1				
TMP ⇐ ALU	13			1	1	1												1			
[RX] ⇐ (TMP)	14	1		1	1					1										11	0

Tabla E.6: Problema 7. Memoria de microprograma

E.4. SECCIÓN DE PROCESAMIENTO: ALGORITMOS ARITMÉTICOS

Muchos de los problemas de este tema son autocomprobables, por lo que se ha omitido su solución.

1. Realizar las siguientes sumas con números de 6 bits: 12+9, 27-15, 14-19, -7-13, 23+10, -20-13; representando los números mediante los siguientes convenios:
   a) Signo y Magnitud
   b) Complemento a 1
   c) Complemento a 2
2. Haz con números de 8 bits trabajando en
   a) Complemento a 1
   b) Complemento a 2

   las operaciones siguientes:
   - $00101101 + 01101111$
   - $11111111 + 11111111$
   - $00000000 - 11111111$
   - $11110111 - 11110111$
3. Encuentra la razón por la que, en el algoritmo de suma/resta en *C1*, hay que sumar el carry de salida al resultado.

   Solución:

   Para hacer la demostración vamos a ir viendo qué ocurre en cada uno de los casos posibles:

   a) Los dos números a sumar son positivos ($x_n = y_n = 0$). Como los dos bits de signo son cero, $c_{n+1} = 0$ siempre. En este caso puede ocurrir:
      1) $s_n = c_n = 0 \rightarrow$ situación normal
      2) $s_n = 1$, como debería ser cero (suma de positivos es positiva), nos indica que existe desbordamiento en la suma

   b) Los dos números a sumar son de distinto signo. Veremos que ocurre si $x > 0$ e $y < 0$. El otro caso es simétrico, por lo que la demostración es similar, intercambiando x por y.

   $$C1(x) + C1(y) = \|x\| + 2^n - 1 - \|y\| = 2^n - 1 + \|x\| - \|y\|$$

   $\|y\| > \|x\| \rightarrow$ el resultado es negativo y queremos que esté en C1, que es lo que tenemos (no aparece acarreo):

$$C1(y-x) = 2^n - 1 - (y-x)$$

$\|x\| > \|y\| \rightarrow$ el resultado es positivo, queremos tener $x-y$ y sin embargo hemos obtenido:

$$2^n - 1 + x - y$$

Esto se traduce, por un lado, en un uno en la posición del c_{n+1} (2^n) y en otro uno que aparece restando y que nos sobra. Para cancelarlos lo que hacemos es sumar el acarreo obtenido con lo que desaparece el acarreo y se compensa el -1.

c) Los dos números a sumar son negativos ($x_n = y_n = 1$). Como los dos bits de signo son uno, $c_{n+1} = 1$ siempre. Si $s_n = 0$ se habrá producido overflow, pues no podemos tener un resultado positivo. En este caso también se debe sumar el acarreo para obtener el resultado correcto:

$$\begin{aligned} C1(x) + C1(y) &= 2^n - 1 - \|x\| + 2^n - 1 - \|y\| \\ &= 2^n - 1 + \underbrace{2^n - 1 - (x+y)} \end{aligned} \qquad \text{(E.1)}$$

Solo nos interesará la parte que abarca la llave como resultado. Para elminar el resto hacemos lo mismo de antes: sumar el acarreo (2^n) con el -1, con lo que se cancelan.

A modo de conclusión, podemos afirmar que siempre que aparezca un acarreo habrá que sumarlo al resultado para corregirlo.

4. Los números decimales con signo de $n-1$ dígitos se pueden representar mediante n dígitos sin signo utilizando la representación en complemento a 9. El complemento a 9 es el complemento a la base menos 1 cuando la base es 10 (igual que el C1 es el complemento a la base menos uno cuando la base es 2). El C9 de un número decimal N se obtiene mediante la siguiente expresión:

$$C9(N) = 10^n - 1 - N$$

donde n es el número de dígitos con que trabajo. Una técnica para hacer el C9 de un número consiste en restar cada dígito de 9. Así, el negativo de 014725 es 985274. Se pide:

a) Expresa como números de tres dígitos en complemento a 9 los siguientes números: 6, -2, 99, -12, -1, 0.

b) Determinar la regla por la cual se suman los números en complemento a 9.

c) Realizar las siguientes sumas con dicha técnica:

1) $0001 + 9999$
2) $0001 + 9998$
3) $9997 + 9996$
4) $9241 + 0802$

Solución:

El modo de proceder es como en complemento a uno cuando trabajamos en binario (ver apuntes de teoría, ya que el complemento a 9 es otro caso particular del complemento a la base menos uno, en el que la base es 10).

5. Los números en complemento a 10 son análogos a los números en complemento a 2. Un número negativo en C10 se forma con sólo sumar 1 al número correspondiente en C9, prescindiendo del acarreo. Más formalmente, el C10 de un número decimal N se obtiene mediante la siguiente expresión:

$$C10(N) = 10^n - N$$

¿Cuál será la regla para la adición en complemento a 10?

Solución:

El modo de proceder es como en complemento a dos cuando trabajamos en binario (ver apuntes de teoría, ya que el complemento a 10 es otro caso particular del complemento a la base, en el que la base es 10).

6. Cuando realizamos operaciones de suma/resta en decimal, realmente estamos utilizando un algoritmo de suma/resta para números decimales representados en signo/magnitud, donde el "dígito" de signo esta representado por el + o el − que antecede al número. Realiza, con aritmética decimal de tres dígitos más el signo, las siguientes sumas y comprueba que el algoritmo que debes utilizar es semejante al visto en teoría para los números en binario: $(-264)+(-858)$, $(+858)+(-264)$, $(+264)+(-858)$.

7. Árbol de Wallace

a) Dibujar el árbol de Wallace para sumar 6 números enteros mediante CSA's (Carry Save Adders). ¿Cómo ha de ser el último sumador?

b) Sumar $12 + 5 + 7 + 3 + 10 + 9$ siguiendo la estructura previa.

Solución:

a) *La arquitectura necesaria para implementar el sumador de 6 sumandos aparece representada en la figura E.3. El último sumador es un sumador de acarreo anticipado o propagado (SAP).*

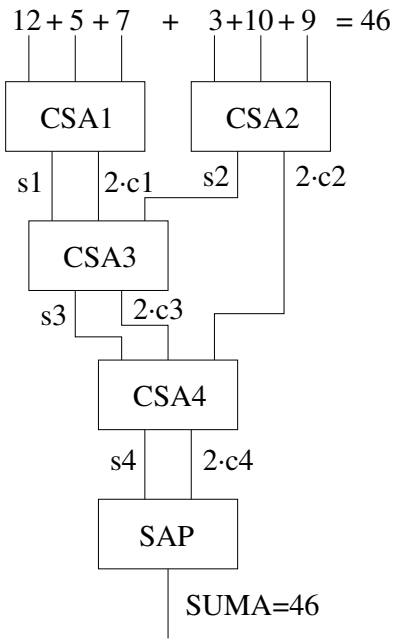


Figura E.3: Árbol de Wallace para seis sumandos

b) En primer lugar sumamos 12+5+7, para conseguir `s1` y `c1`. En paralelo sumamos 3+10+9 obteniendo `s2` y `c2`; a continuación sumamos `s1+2c1+s2` seguido de `s3+2c3+2c2`. Por último en un sumador convencional sumamos `s4+2c4`.

```
001100 x1          000011 x4
000101 x2          001010 x5
000111 x3          001001 x6
---------          ---------
001110 s1          000000 s2
000101 c1          001011 c2

      s1   001110
     2c1   001010
      s2   ------
           000100 s3
           001010 c3
```

```
 s3   000100
2c3   010100
2c2   010110
      ------
      000110 s4
      010100 c4

 s4   000110
2c4   101000
      ------
      101110 = 46
```

Deberemos usar sumadores de al menos 6 bits (ó siete si trabajamos en C2) para evitar desbordamiento de la suma.

8. ¿Cuándo tiene sentido utilizar sumadores con acarreo almacenado (CSA)? Construir el Árbol de Wallace para sumar $7 - 5 + 9$ y realizar la suma utilizando aritmética de 6 bits en C2.

   Solución:

   Los sumadores de accarreo almacenado son especialmente útiles cuando estamos trabajando con más de dos operandos. En la figura E.4 aparece representado el árbol pedido.

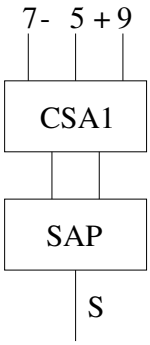


Figura E.4: Árbol de Wallace para tres sumandos

```
000111 x1
111011 x2
001001 x3
------
110101  s
001011  c
```

```
110101  s
010110 2c
------
001011 = 11 (once en decimal)
```

9. Realiza las siguientes multiplicaciones mediante el algoritmo de multiplicación binario de suma y desplazamiento con n=5.
   a) 12×10
   b) 7×12
   c) 6×15
10. Realiza las siguientes divisiones mediante el algoritmo de división binario con restauración con n=5.
   a) $12 \div 3$
   b) $15 \div 2$
11. Realiza las siguientes operaciones con números en punto flotante en el formato IEEE 754.
   a) $C30C0000 + C1500000$
   b) $3B370000 + 39F68000$

   Solución:

   a) $C30C0000 \rightarrow 1 \mid 100\ 0011\ 0 \mid 000\ 1100\ 00..00 = -1.00011 \times 2^7$
   $C1500000 \rightarrow 1 \mid 100\ 0001\ 0 \mid 101\ 0000\ 00..00 = -1.101 \times 2^3$

   Igualamos los exponentes al mayor de los dos y sumamos. El resultado obtenido se vuelve a normalizar.

$$\begin{array}{r} -1.000\ 1100 \times 2^7 \\ \underline{-0.000\ 1101 \times 2^7} \\ -1.001\ 1001 \times 2^7 \end{array}$$

   $-1.001\ 1001 \times 2^7 \rightarrow C3190000$

   b) $3B370000 \rightarrow 0 \mid 011\ 1101\ 0 \mid 011\ 0111\ 00..00 = 1.0110111 \times 2^{-9}$
   $39F68000 \rightarrow 0 \mid 011\ 1001\ 1 \mid 111\ 0110\ 1000\ 00..00 = 1.11101101 \times 2^{-12}$

$$\begin{array}{r} 1.01101110000 \times 2^{-9} \\ \underline{0.00111101101 \times 2^{-9}} \\ 1.10101011101 \times 2^{-9} \end{array}$$

   $1.10101011101 \times 2^{-9} \rightarrow 3B55D000$

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Bibliografía

[1] A.S.Tanembaum: "Structured Computer Organization", Cuarta edición, Prentice-Hall, 1999.

[2] D.A.Patterson, J.L.Hennessy: "Organización de Computadores", Mc. Graw-Hill, 1995.

[3] D.A.Patterson, J.L.Hennessy: "Computer Organization & Design. The hardware/software interface", Segunda edición. Morgan Kaufmann Publishers, 1998.

[4] Pedro de Miguel Anasagasti: "Fundamentos de los Computadores", Paraninfo, Tercera edición, 1992.

[5] A.S.Tanembaum: "Organización de Computadoras", Segunda Edición, Prentice-Hall, 1986.

[6] J.P.Hayes: "Computer Architecture and Organization", Segunda Edicion, Mc. Graw-Hill, 1988.

[7] P.de Miguel, J.M.Angulo: "Arquitectura de Computadores", Paraninfo, 1987.

[8] M. Morris Mano: "Arquitectura de Computadores", Prentice Hall, 1983.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.

Parece que ya deja de sorprender la creciente omnipresencia de los ordenadores en todos los ámbitos de la sociedad. A nosotros nos sigue impresionando la rapidísima evolución de los sistemas basados en computador, su creciente potencia de cálculo capaz de resolver cada vez problemas de mayor complejidad y su capacidad de simplificar y reducir el tiempo necesario para realizar muchas tareas cotidianas. Pues bien, los fundamentos, conceptos y modos de operación de estas máquinas tan comunes hoy en día, son los que tratamos de introducir y desentrañar en este texto. O con otras palabras, este libro está orientado a aquellas personas que alguna vez se han preguntado "¿Cómo es posible que los transistores y puertas lógicas que hay dentro de mi ordenador me permitan editar un archivo o ejecutar un programa que he escrito en Modula o en C?", pregunta, que por otro lado, esperamos se hayan planteado todos nuestros alumnos de asignaturas de introducción a los computadores.

Aunque no son del todo necesarios, suponemos que el lector tiene algunos conocimientos de electrónica digital y programación. Pues bién, en este libro precisamente queremos cubrir el desnivel semántico que existe en un sistema computador entre esas dos materias (electrónica digital y lenguajes de alto nivel), contemplando el control microprogramado y cableado, el lenguaje ensamblador y los sistemas operativos, según desglosamos a continuación por temas.

El tema 1 introduce los primeros conceptos básicos y una descripción inicial de la arquitectura de von Neumann. El tema 2 describe los convenios comúnmente utilizados para representar números, caracteres e instrucciones en el computador. A partir del tema 3 profundizamos en la arquitectura de computadores siguiendo un esquema estructural, en el que el computador se compone del procesador (tema 3), el cual engloba la sección de control (tema 4) y de procesamiento (tema 5), jerarquía de memoria (tema 6) y unidad de Entrada/Salida (tema 7). Por último, el tema 8 describe como los Sistemas Operativos permiten gestionar toda la arquitectura, dando una visión global del computador.

R. Asenjo, E. Gutierrez, J. Ramos. DAC-UMA.