ESTRUCTURA Y TECNOLOGÍA DE COMPUTADORES

Sebastián Dormido
Mª Antonia Canto
José Mira
Ana E. Delgado

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

SANZ Y TORRES

Contenido

Prólogo ..... **xvii**

1 Estructuras de interconexión de un computador **1**

1.1 Componentes de un computador 2
1.2 Función de un computador 6
1.2.1 Ciclos de búsqueda y ejecución 6
1.2.2 Ejemplo: Ejecución de una instrucción 8
1.2.3 Diagrama de flujo del ciclo de instrucción 10
1.2.4 El ciclo de interrupción 12
1.2.5 Función de E/S 15
1.3 Estructuras de interconexión 16
1.4 Interconexión mediante bus 20
1.4.1 Diagramas de temporización 21
1.4.2 Estructura de bus 22
1.4.3 Estructura jerárquica de buses 24
1.4.4 Elementos de diseño del bus 26
1.4.5 Consideraciones prácticas en la conexión mediante bus 31
1.4.6 Ejemplo de estructura de bus: El Unibus 36
1.5 Conclusiones 41
1.6 Problemas 42

2 Unidad de memoria **45**

2.1 Definiciones y conceptos básicos 46
2.1.1 Localización 46
2.1.2 Capacidad 47
2.1.3 Unidad de transferencia 47
2.1.4 Método de acceso 47
2.1.5 Tipos físicos 49

2.1.6 Características físicas 49
2.1.7 Velocidad 50
2.1.8 Organización 52
2.1.9 Resumen de características y propiedades de la memoria 52
2.2 Jerarquía de memorias 53
2.2.1 Ejemplo: Sistema con dos niveles de memoria 55
2.3 Memorias de semiconductor 58
2.3.1 Características generales de un CIM 58
2.3.2 Ejemplo: Cálculo del número de ciclos de reloj en los accesos a memoria 60
2.3.3 Estructura de la celda básica de memoria 60
2.3.4 Organización interna 62
2.3.5 Diseño de bloques de memoria 68
2.3.6 Conexión de la unidad de memoria al bus del sistema 71
2.3.7 Estructura y direccionamiento de la unidad de memoria 72
2.4 Memorias caché 75
2.4.1 Rendimiento de una memoria caché 79
2.4.2 Capacidad de la memoria caché 80
2.4.3 Organización de la memoria caché 80
2.4.4 Algoritmos de reemplazamiento 90
2.4.5 Estrategia de escritura 92
2.4.6 Tamaño del bloque 93
2.4.7 Número de cachés 93
2.5 Memorias asociativas 96
2.5.1 Ejemplo: Concepto de memoria asociativa 96
2.5.2 Estructura de una memoria asociativa 96
2.5.3 Determinación de la función lógica del registro de máscara 99
2.5.4 Operación de lectura 100
2.5.5 Operación de escritura 100
2.5.6 Ejemplo: Diseño de una memoria asociativa 102
2.6 Memorias compartidas 103
2.6.1 Asignación de la menor prioridad al elemento servido 104
2.6.2 Rotación de prioridades 105
2.7 Memorias tipo pila 106
2.7.1 Aplicaciones de las memorias tipo pila 110
2.8 Discos magnéticos 112
2.8.1 Estructura física 112
2.8.2 Ejemplo: Tiempo de acceso a un archivo de acceso secuencial y aleatorio 115
2.8.3 Controlador del disco 115
2.8.4 Planificación del disco 118
2.9 Conclusiones 123
2.10 Problemas 124

3 Unidad de entrada-salida 127

3.1 Dispositivos externos 129

3.2 Controlador de E/S 131
3.2.1 Funciones del controlador de E/S 131
3.2.2 Estructura del controlador de E/S 133
3.2.3 Estructura del sistema de E/S 134
3.3 E/S controlada por programa 136
3.3.1 Órdenes de E/S 137
3.3.2 Instrucciones de E/S 138
3.3.3 Ejemplo: Transferencia de E/S controlada por programa en el 68000 140
3.4 E/S por interrupciones 141
3.4.1 Clasificación de las interrupciones 145
3.4.2 Origen de las interrupciones 146
3.4.3 Número de líneas de interrupción 147
3.4.4 Control de la CPU sobre la interrupción 148
3.4.5 Identificación de la fuente de la interrupción y gestión de su prioridad 149
3.4.6 Niveles de interrupción 155
3.4.7 Controlador de interrupciones 158
3.4.8 Ejemplos de controladores de interrupciones 158
3.4.9 Estructura de interrupciones del 68000 162
3.5 Acceso directo a memoria (DMA) 164
3.5.1 Controlador de DMA 164
3.5.2 Transferencia de datos mediante DMA 166
3.5.3 Configuración del DMA 168
3.6 Procesador de E/S (PE/S) 170
3.6.1 Características de los PE/S 171
3.6.2 Clasificación de los PE/S 172
3.7 Conclusiones 174
3.8 Problemas 175

4 Unidad aritmético-lógica 181

4.1 Sumadores binarios 182
4.1.1 Semisumador binario (SSB) 183
4.1.2 Sumador binario completo (SBC) 184
4.1.3 Sumador binario serie 187
4.1.4 Sumador binario paralelo con propagación del arrastre 188
4.1.5 Sumador-restador binario paralelo con propagación del arrastre 189
4.2 Sumadores de alta velocidad 192
4.2.1 Características de los arrastres 193
4.2.2 Sumadores con anticipación del arrastre 193
4.2.3 Sumadores de suma condicional 197
4.2.4 Sumadores con selección del arrastre 199
4.2.5 Sumadores con detección de la finalización del arrastre 199
4.2.6 Sumadores que minimizan el número de arrastres 202
4.2.7 Sumadores con arrastre almacenado 203
4.3 Sumadores en código BCD 207

4.3.1 Organización de los sumadores en código BCD 209
4.3.2 Restador en código BCD 210
4.4 Multiplicadores binarios 212
4.4.1 Multiplicación de "lápiz y papel" de números sin signo 212
4.4.2 Mejoras en el algoritmo de "lápiz y papel" 214
4.4.3 Multiplicación en complemento a 2: Algoritmo de Booth 216
4.4.4 Ejemplo: Algoritmo de Booth 221
4.5 Multiplicadores de alta velocidad 222
4.5.1 Suma rápida de los productos parciales 222
4.5.2 Algoritmo de Booth modificado 225
4.6 Divisores binarios 228
4.6.1 Ejemplo: División de dos números enteros binarios sin signo 228
4.6.2 División por el método de restauración 230
4.6.3 Ejemplo: Algoritmo de restauración 230
4.6.4 División por el método de no restauración 232
4.6.5 Ejemplo: Algoritmo de no restauración 233
4.7 Estructura de la unidad aritmético-lógica (ALU) 234
4.7.1 ALU´s integradas 235
4.8 Aritmética en coma flotante 238
4.8.1 La representación de números en coma flotante 239
4.8.2 La normalización de números en coma flotante 239
4.8.3 Exponentes polarizados 241
4.8.4 Posibles sistemas en coma flotante 243
4.8.5 El formato IEEE de representación de números en coma flotante 244
4.8.6 Ejemplo: Representación en el formato IEEE 245
4.8.7 Operaciones aritméticas en coma flotante 246
4.8.8 Algoritmo de suma y resta en coma flotante 247
4.8.9 Redondeo y truncamiento 249
4.8.10 Algoritmo de multiplicación y división en coma flotante 250
4.8.11 Estructura básica de una unidad aritmética en coma flotante 253
4.9 Operaciones de desplazamiento 255
4.9.1 Clasificación de las operaciones de desplazamiento 255
4.9.2 Ejemplo: Diseño de un registro de desplazamiento de 4 bits 256
4.9.3 Estructura de los registros de desplazamiento 258
4.10 Operaciones de comparación 261
4.10.1 Utilizando un circuito combinacional 261
4.10.2 Utilizando un circuito secuencial 263
4.10.3 Utilizando un sumador 263
4.11 Conclusiones 266
4.12 Problemas 267

5 Diseño de transferencia entre registros 271

5.1 Niveles de diseño de un sistema digital: diseño jerárquico 274
5.2 Nivel de transferencia entre registros 276

5.2.1 Representación 276
5.2.2 Expandibilidad de los componentes 278
5.3 Estructura de un sistema digital 279
5.3.1 Componentes de un sistema digital 280
5.3.2 Puntos de control 281
5.3.3 Modelo de Glushkov 283
5.4 Diagramas de máquinas de estados algorítmicas (ASM) 286
5.5 Ejemplo de diseño: multiplicador binario 289
5.5.1 Multiplicador binario 289
5.5.2 Unidad de procesamiento o ruta de datos del multiplicador 291
5.5.3 Diagrama ASM del multiplicador 291
5.5.4 Unidad de control con lógica cableada 294
5.5.5 Unidad de control diseñada con elementos de memoria tipo D 297
5.5.6 Unidad de control diseñada con un registro de secuencia y un decodificador 300
5.5.7 Unidad de control diseñada con un elemento de memoria por estado 304
5.5.8 Unidad de control diseñada utilizando un registro de estado y una memoria ROM 307
5.5.9 Unidad de control diseñada utilizando un contador y un decodificador 312
5.5.10 Unidad de control diseñada utilizando un registro de estado y un PLA 315
5.5.11 Resumen del procedimiento de diseño a nivel de registro 318
5.6 Conclusiones 319
5.7 Problemas 320

6 Diseño del procesador 327

6.1 Repertorio de instrucciones 331
6.1.1 Procesadores de tres direcciones 333
6.1.2 Procesadores de dos direcciones 335
6.1.3 Procesadores de una dirección (procesadores con acumulador) 336
6.1.4 Procesadores de cero direcciones (procesadores con pila) 337
6.1.5 Procesadores sin ALU 340
6.1.6 Análisis de las diferentes arquitecturas de procesadores 340
6.1.7 Procesadores con banco de registros 341
6.1.8 Arquitectura de carga/almacenamiento: Procesadores RISC 342
6.2 Modos de direccionamiento 345
6.3 Ciclo de ejecución de una instrucción 350
6.3.1 Fase de búsqueda de la instrucción 350
6.3.2 Fase de decodificación de la instrucción 353
6.3.3 Fase de búsqueda de los operandos 353
6.3.4 Fase de ejecución de la instrucción 354
6.3.5 Transferencia a un subprograma 355
6.3.6 Ciclo de interrupción 356
6.4 Fases en el diseño del procesador 360
6.5 Diseño de un procesador elemental 361
6.5.1 Especificación del procesador SIMPLE1 359
6.5.2 Repertorio de instrucciones 362

6.5.3 Diagrama de flujo del repertorio de instrucciones 363
6.5.4 Asignación de recursos a la unidad de procesamiento o ruta de datos 364
6.5.5 Obtención del diagrama ASM del procesador 365
6.5.6 Diseño de la unidad de control 366
6.5.7 Diseño de la unidad de procesamiento o ruta de datos 374
6.6 Conclusiones 376
6.7 Problemas 377

7 Microprogramación 383

7.1 Modelo original de Wilkes 385
7.2 Estructura de una unidad de control microprogramada 387
7.2.1 Conceptos básicos 387
7.2.2 Elementos de una unidad de control microprogramada 387
7.2.3 Secuenciamiento de las microinstrucciones 390
7.2.4 Organización de la memoria de control 395
7.2.5 Ejecución de las microinstrucciones 399
7.3 Representación de los microprogramas 401
7.3.1 Ejemplo: Desarrollo de un sencillo microprograma 404
7.4 Ejemplo de diseño microprogramado: multiplicador binario 412
7.5 Unidad de control microprogramada de un computador 418
7.5.1 Estructura de SIMPLE2 418
7.5.2 Formato de la microinstrucción de *ucμP* 423
7.5.3 Fases en la ejecución de una microinstrucción 424
7.5.4 Inicialización de SIMPLE2 425
7.5.5 Búsqueda y decodificación de las instrucciones máquina 425
7.5.6 Ejemplo: Microprogramación de un repertorio de instrucciones máquina de SIMPLE2 .428
7.6 Conclusiones 432
7.7 Problemas 434

A Componentes combinacionales 439

A.1 Puertas de palabras 440
A.2 Codificadores 441
A.2.1 Codificadores con prioridad 442
A.3 Decodificadores 446
A.4 Multiplexores 451
A.5 Demultiplexores 454
A.6 Dispositivos lógicos programables 456
A.6.1 Arrays lógicos programables (PLA) 456
A.6.2 Ejemplo: Síntesis de funciones lógicas con un PLA 459
A.6.3 Arrays lógicos programables (PAL) 459
A.6.4 Ejemplo: Síntesis de funciones lógicas con un PAL 461

B Introducción a los circuitos secuenciales 463

B.1 Concepto de máquina secuencial 465
B.2 Sistemas síncronos y asíncronos 467
B.2.1 Estructura canónica de una máquina secuencial 468
B.2.2 Diagrama de estados y tabla de estados 469
B.2.3 Minimización del número de estados 470
B.2.4 Codificación binaria y el problema de la asignación de estados 474
B.3 Elemento de memoria 477
B.4 Tipos de elementos de memoria 482
B.4.1 Elemento de memoria RS 482
B.4.2 Elemento de memoria JK 482
B.4.3 Elemento de memoria D 484
B.4.4 Elemento de memoria T 484
B.5 Elemento de memoria maestro-esclavo 480
B.6 Análisis de circuitos secuenciales 490
B.6.1 Circuito secuencial realizado en forma canónica 490
B.6.2 Ejemplo 1: Análisis de un circuito secuencial síncrono 490
B.6.3 Circuito secuencial realizado con elementos de memoria 490
B.6.4 Ejemplo 2: Análisis de un circuito secuencial síncrono 492
B.7 Síntesis de circuitos secuenciales 495
B.7.1 Modelar las especificaciones 495
B.7.2 Minimizar el número de estados 496
B.7.3 Codificar en binario las entradas, salidas y estado 496
B.7.4 Ecuaciones de entrada a los elementos de memoria 496
B.7.5 Ejemplo 1: Detector de secuencia mediante un modelo de Huffman-Mealy 497
B.7.6 Ejemplo 2: Detector de secuencia mediante un modelo de Moore 501
B.7.7 Ejemplo 3: Complementador a 2 504
B.7.8 Ejemplo 4: Contador binario módulo-8 505
B.7.9 Ejemplo 5: Contador de código arbitrario con dos secuencias 508
B.7.10 Diagramas de estado que utilizan expresiones lógicas para expresar las transiciones 510
B.7.11 Ejemplo 6: Generador de pulso con anchura de pulso variable 511
B.7.12 Tabla de estados con expresiones de transición 513
B.8 Síntesis de circuitos secuenciales con PLA´s o PAL´s 515
B.8.1 Dispositivos lógicos programables con registros 515
B.8.2 Ejemplo: Diseño de un contador con un PAL 516
B.8.3 Macroceldas 518

Bibliografía 521

Índice 523

Prólogo

Objetivos

El objetivo básico del libro es dar una visión, lo más completa posible, de los fundamentos de la arquitectura, organización y diseño de computadores.

La *arquitectura de computadores* estudia la estructura y comportamiento de los diferentes módulos funcionales de un computador y como interaccionan entre sí para proporcionar las necesidades de procesamiento de los usuarios. Es decir, su finalidad es considerar los atributos de un computador que son visibles a un programador a nivel de lenguaje máquina. Ejemplos de estos atributos son el repertorio de instrucciones, los mecanismos de E/S y las técnicas de direccionamiento de memoria.

La *organización de los computadores* se ocupa de la descripción de sus unidades operacionales y de como se conectan para obtener una arquitectura dada. Su interés se centra en cuestiones, que son transparentes al usuario de un computador, tales como: las interfaces entre el computador y los periféricos, la tecnología de memoria y las señales de control utilizadas, etc.

Un ejemplo que ilustra la distinción entre arquitectura y organización de un computador es el siguiente: El diseñador de la arquitectura tiene que decidir, por ejemplo, si se va a incluir dentro del repertorio de instrucciones una de dividir. La forma de realizar esta instrucción, si es utilizando una unidad aritmético-lógica que incorpora la división o mediante un algoritmo que utiliza otros operadores aritméticos, es materia de la organización.

El diseño de un computador estudia la realización física de sus circuitos lógicos tomando en consideración un conjunto de especificaciones dadas.

A quien va dirigido este texto

Estructura y Tecnología de Computadores está concebido como texto base de la asignatura *Estructura y Tecnología de Computadores II* que tienen que cursar los alumnos de la Escuela de Informática de la UNED, en el 1er curso de las carreras de Ingeniero Técnico en Informática de Sistemas

y de Ingeniero Técnico en Informática de Gestión. Este hecho impone una serie de ligaduras sobre la estructuración de sus contenidos y la forma de desarrollarlos ya que tienen que ser, en la medida de lo posible, autosuficientes y comprensibles a estudiantes que no asisten de forma habitual a clases presenciales.

Esta reflexión ha llevado a los autores a incluir dos apéndices en los que se introducen los principales componentes combinacionales que se emplean en la descripción de un computador y los circuitos secuenciales cuya comprensión es fundamental para poder entender el funcionamiento de la unidad de control de un computador. Los alumnos que cursan Ingeniería Técnica en Informática de Sistemas ven estos conceptos con mayor profundidad en la asignatura de *Electrónica Digital*; no ocurre lo mismo con los alumnos de Ingeniería Técnica en Informática de Gestión que no los estudian en ninguna otra asignatura y sin embargo los necesitan para la comprensión de este texto. La solución que se ha adoptado es incluir, aún a costa de una mayor extensión, los dos apéndices mencionados.

Tambien puede resultar útil a estudiantes de Facultades de Ciencias o Escuelas de Ingenierías que tengan en sus planes de estudio materias relacionadas con la Arquitectura de Computadores.

Organización del texto

El texto se ha estructurado en 7 temas y 2 apéndices cuyos contenidos son:

Tema 1: Estructuras de interconexión de un computador

En este tema se analizan las estructuras básicas utilizadas para la interconexión de los elementos de un computador. Como introducción se hace una breve revisión de los componentes fundamentales de la arquitectura clásica de von Neumann. Se consideran los principales enfoques estructurales para la interconexión, incluyendo consideraciones temporales y los protocolos de arbitraje en el intercambio de información entre componentes. Por su particular interés e importancia se dedica una atención especial a la estructura de interconexión tipo bus.

Tema 2: Unidad de memoria

El objetivo de este tema es el estudio del almacenamiento de información de un computador que contiene tanto los programas que ejecuta como los datos sobre los que han de actuar dichos programas. Un computador contiene una *jerarquía de subsistemas de memoria*, unos internos al propio computador (directamente accesibles desde la CPU) y otros externos (accesibles desde la CPU a través de un módulo de E/S).

En este tema se estudian, en primer lugar, los conceptos básicos y las características de los diferentes tipos de memoria. En segundo lugar se analiza el compromiso que establecen los parámetros de capacidad, velocidad y coste en la jerarquía de memorias y cómo el principio de localidad tanto espacial como temporal permite organizar los datos de una forma eficaz en los diferentes niveles de memoria.

De forma más concreta se estudian las memorias de tipo semiconductor que utilizan como componente básico los circuitos integrados de memoria, para pasar después a analizar su utilización como componentes del computador. Se presenta de forma detallada un elemento esencial de todos los computadores actuales: la *memoria caché*. El tema finaliza estudiando otras arquitecturas de sistemas de memoria como son las memorias de carácter asociativo, la utilización de memoria compartida, el concepto de memorias de tipo pila y una introducción a los discos magnéticos que son uno de los principales sistemas de almacenamiento secundario en los computadores de hoy día.

Tema 3: Unidad de entrada-salida

Además de la CPU y de la memoria el tercer elemento clave de un computador es la unidad de entrada/salida (E/S). La unidad de E/S proporciona un método de comunicación eficaz entre el sistema central y el usuario. Un computador basa gran parte de su utilidad en la capacidad de recibir y transmitir información desde o hacia el exterior.

Se puede ver la E/S como un conjunto de controladores de E/S; cada controlador se conecta al bus del sistema o a un conmutador centralizado y supervisa a uno o más dispositivos periféricos. Un controlador de E/S no es simplemente una conexión física entre un dispositivo externo y el bus del sistema, sino que contiene cierta "inteligencia". Es decir, dispone de la lógica necesaria para realizar una función de comunicación entre los periféricos del computador y el bus.

Se introducen los dispositivos externos, mostrándose en términos muy generales su naturaleza. La comunicación de los dispositivos externos con el controlador de E/S pone en juego diferentes categorías de señales: datos, control y estado del dispositivo. En este tema se presenta una visión unificada de la estructura y función de un controlador de E/S. El núcleo central lo constituye el estudio de los mecanismos por los que un controlador de E/S interacciona con el resto del computador. Se utilizan las siguientes técnicas:

a) *E/S controlada por programa*. La CPU ejecuta un programa que tiene el control completo y directo de la operación de E/S. Cuando la CPU transfiere una orden al controlador de E/S debe esperar hasta que acabe dicha operación.

b) *E/S por interrupciones*. La CPU envía una orden de E/S y continúa ejecutando otras instrucciones hasta que es interrumpida por el controlador de E/S cuando ha finalizado su tarea. La CPU no malgasta ahora su tiempo esperando a que se realice una operación de E/S, lo que aumenta el rendimiento global del sistema.

c) *Acceso directo a memoria* (DMA). Con esta técnica al controlador de E/S se le permite el acceso directo a memoria a través del módulo DMA. Se pueden transferir bloques de datos a o desde memoria sin intervención directa de la CPU, excepto en los instantes iniciales y finales de la transferencia.

El tema finaliza con la introducción del concepto de procesador de E/S (PE/S). El PE/S es una extensión natural del acceso directo a memoria. Un PE/S tiene la capacidad de ejecutar instrucciones de E/S lo que le da un control completo sobre dicha operación. La CPU no ejecuta ahora las

instrucciones de E/S, que se almacenan en memoria principal para ejecutarse en un PE/S. La CPU comienza una transferencia de E/S al enviar una orden al PE/S para que ejecute el programa correspondiente. Se trata pues de un potente controlador de E/S capaz de ejecutar un programa.

Tema 4: Unidad aritmético-lógica

La *unidad aritmético-lógica* (ALU) es la parte del computador donde se efectúan las operaciones aritméticas y lógicas sobre los datos. Las otras unidades del computador (unidad de control, memoria y unidad de E/S) son las encargadas de suministrar datos a la entrada de la ALU y recibirlos nuevamente una vez procesados.

Los datos llegan a la ALU a través de registros y los resultados que se generan también se almacenan en registros. Estos registros son memorias temporales dentro de la CPU que se conectan mediante el bus de datos con la ALU. Cuando la ALU finaliza una operación, activa determinados indicadores que pueden ser utilizados por la unidad de control. La unidad de control envía señales que controlan las operaciones y el movimiento de datos de entrada y salida de la ALU. En este tema se estudian los algoritmos y los circuitos asociados que realizan las cuatro operaciones aritméticas básicas, tanto en coma fija como en coma flotante. Un número en coma flotante está constituido por un par de números en coma fija, la mantisa m y el exponente e y se utiliza para representar números de la forma $m \times B^e$, donde B es la base que está implícita. La representación en coma flotante aumenta el rango de los números que se pueden expresar para una longitud de palabra dada, aunque requieren circuitos aritméticos mucho más complejos que cuando se emplea coma fija. Con el fin de proporcionar una representación única para cada número en coma flotante se realiza un proceso de normalización. También se analizan las operaciones de desplazamiento y de comparación.

Tema 5: Diseño de transferencia entre registros

En este tema se estudia el procedimiento de diseño de un computador o con mayor generalidad de cualquier sistema digital. El método de diseño puede considerarse a diferentes niveles de detalle o complejidad. Se suelen emplear los tres niveles siguientes:

1) *Nivel de puertas lógicas.* Corresponde al estudio de la teoría de la conmutación y tiene interés para los diseñadores de sistemas. En este nivel la unidad de información es el **bit**.
2) *Nivel de registros.* Un programador en lenguaje máquina o lenguaje ensamblador contempla a un sistema digital desde este nivel. No se tratan ya las puertas lógicas de forma individual, sino los registros de la máquina y sus interconexiones. Es el nivel que se emplea cuando se consideran las características estructurales de un sistema digital. La unidad de información es la **palabra**.
3) *Nivel de procesador.* Los objetos básicos que se consideran en este nivel son elementos tales como CPU, procesadores de E/S, memorias etc. La característica fundamental de estos módulos es su elevada complejidad. La unidad de información es un **bloque de palabras** que puede representar o un programa o una zona de datos.

Se presenta una metodología general de diseño de sistemas digitales cuando se describen a nivel de transferencia de registros. Se exponen los fundamentos de los lenguajes de descripción que se pueden utilizar tanto en los niveles de especificación como de realización. La descripción de un sistema digital se divide en dos partes: la *unidad de procesamiento* y la *unidad de control* (modelo de Glushkov).

La *unidad de procesamiento* es la parte del sistema digital en la que se almacenan y transforman los datos. La función de la *unidad de control* es generar una secuencia de señales de control de acuerdo con el algoritmo de transferencia de registros, que especifica la realización de la operación deseada. En muchos casos, una máquina secuencial es un modelo adecuado para la realización de la unidad de control. En este tema se presenta una forma de carácter gráfico, de especificar el modelo de un sistema secuencial: *el diagrama de máquina de estados algorítmica* o *diagrama ASM* (acrónimo de Algorithmic State Machines). El diagrama ASM se caracteriza por el hecho de que describe una secuencia de sucesos así como la relación temporal entre los estados de la unidad de control y las acciones que ocurren en los estados como respuesta a los distintos pulsos de reloj.

Se analiza con detalle la organización de la unidad de control y se describen diferentes realizaciones de la misma. Con el fin de reforzar los conceptos que se van introduciendo a lo largo del tema y a modo de ejemplo ilustrativo se desarrolla de forma detallada la realización de un multiplicador de dos números binarios sin signo de n bits, basado en el algoritmo de *lápiz y papel mejorado* propuesto en el tema 4.

Tema 6: Diseño del procesador

La unidad central de proceso (CPU) es un elemento esencial de cualquier computador ya que tiene como misión ejecutar las instrucciones de un programa. Las instrucciones se procesan de forma secuencial, leyéndose de posiciones consecutivas de memoria (tras ejecutar la instrucción que se encuentra en la dirección x se ejecuta la instrucción que está en la dirección $x+1$ y así sucesivamente). Este esquema que se conoce como *secuenciamiento implícito* sólo puede ser modificado mediante instrucciones de bifurcación o salto. En este tipo de instrucciones se debe especificar la dirección de la siguiente instrucción que se va a ejecutar o un mecanismo para calcularla.

En este tema se estudia la unidad de control de un computador realizada con lógica cableada (se construye con puertas lógicas y su concepción obedece a las técnicas clásicas de diseño de circuitos secuenciales). En primer lugar se analiza el *repertorio de instrucciones* de los procesadores y se procede a una clasificación de los mismos según el número de direcciones que contenga el formato de sus instrucciones. Se describen los *modos de direccionamiento* de un computador que permiten calcular de forma no ambigua la dirección real o efectiva donde se encuentran los operandos de la instrucción y dónde hay que almacenar el resultado que produce.

Se analiza de forma detallada el *ciclo de ejecución de una instrucción*. La ejecución de una instrucción siempre conlleva realizar la misma secuencia de pasos independientemente del repertorio de instrucciones específicas que posea el procesador y de los campos y modos de direccionamiento que se hayan definido. De forma genérica, el ciclo de ejecución de una instrucción se puede subdividir

en cuatro fases: 1) Fase de búsqueda de la instrucción, 2) Fase de decodificación de la instrucción, 3) Fase de búsqueda de los operandos y 4) Fase de ejecución de la instrucción.

Se presentan también las etapas que son necesarias para realizar de una forma sistemática el diseño de un procesador genérico con lógica cableada. Finalmente en la última sección del tema se procede a realizar el diseño de SIMPLE1 que es un procesador elemental y académico que permite mostrar de forma sencilla los conceptos que se han ido introduciendo. El problema se especifica dando las características estructurales de SIMPLE1, un procesador ficticio extraordinariamente simplificado, cuya única finalidad es mostrar paso a paso su diseño con lógica cableada.

Tema 7: Microprogramación

La microprogramación se puede considerar, según la idea original expuesta por el prof. M. V. Wilkes de la Universidad de Cambridge en 1950, *como un método sistemático para diseñar la unidad de control de cualquier sistema digital*. Sin embargo esta primera concepción de la microprogramación como un procedimiento diferente al tradicionalmente utilizado para el diseño de la unidad de control con lógica cableada, ha evolucionado llegando a convertirse en una alternativa no sólo de diseño sino de realización.

En la década de los 50 el concepto de la microprogramación sólo generó un relativo interés desde el punto de vista teórico. El principal motivo para este hecho estuvo en el alto coste que suponía su realización debido a la limitada tecnología de las memorias disponibles. En la mitad de la década de los 60 se produce un hecho fundamental para el desarrollo de la microprogramación, cuando el fabricante de computadores IBM la utiliza en la mayor parte de sus modelos de la *serie 360*, para el diseño de la unidad de control.

En este tema se presentan los fundamentos de la microprogramación. En primer lugar se introduce el modelo original propuesto por M. Wilkes. A continuación se presentan las diferentes alternativas que se han propuesto para el diseño de una unidad de control microprogramada. El tema finaliza con dos ejemplos de diseño de unidad de control microprogramada: el multiplicador binario analizado en el tema 5 y SIMPLE2 que es un sencillo computador que dispone de una unidad de control microprogramable (*ucμP*).

Apéndice A: Componentes combinacionales

Un *circuito combinacional* es un circuito lógico cuyos valores de salida están determinados en cualquier instante de tiempo únicamente por los valores aplicados a sus entradas, y por tanto son independientes de los estados anteriores de las mismas. Los circuitos combinacionales no nos permiten pues almacenar el estado de las entradas y utilizarlas posteriormente para tomar decisiones, es decir, son circuitos que no tienen memoria.

La característica principal de un circuito combinacional es la función lógica que realiza. Esta función lógica describe el comportamiento del circuito pero no tiene por qué describir su estructura

interna, por lo que es posible que existan diferentes circuitos lógicos que realizan la misma función. Los circuitos combinacionales tienen dos usos principales en los sistemas digitales:

a) *Transferencia de datos*. Controlan el flujo de señales lógicas de una parte del sistema a otra.

b) *Procesamiento de datos*. Transforman los datos realizando los cálculos necesarios.

En este apéndice se consideran los principales componentes combinacionales que se emplean en la descripción a nivel de registro de un sistema digital: puertas de palabras, codificadores, decodificadores, multiplexores, demultiplexores y dispositivos lógicos programables.

Aunque algunos de estos componentes ya se han introducido en los propios temas se exponen de forma conjunta con la finalidad de hacer el texto lo más autocontenido posible. No se consideran los circuitos sumadores, multiplicadores, comparadores y unidades aritmético-lógicas ya que a su estudio exclusivo se dedicó el tema 4.

Apéndice B: Introducción a los circuitos secuenciales

Los circuitos combinacionales no nos permiten por si mismos resolver el problema de almacenar el valor que tienen las entradas en un instante y utilizarlas posteriormente para tomar decisiones cuando modifiquen sus valores.

En un circuito combinacional el valor de sus salidas depende únicamente del valor que hay presente en sus entradas. Por el contrario *en un circuito secuencial las salidas en un instante cualquiera t depende no sólo del valor actual de sus entradas sino también de sus valores previos*. Así pues, un circuito secuencial debe recordar cierta información acerca de los valores pasados de sus entradas. Esta propiedad da una gran generalidad a este tipo de circuitos y permite generar respuestas mucho más interesantes y complejas que las que produce un circuito combinacional.

La unidad de control de un computador es un ejemplo de un circuito secuencial sofisticado, pero también son circuitos secuenciales dispositivos más sencillos como una máquina de expedición automática de billetes de tren. En una máquina de este tipo se introducen secuencialmente los billetes o monedas y el resultado de la operación no depende sólo de lo último que se ha insertado sino también de lo previo. Con este amplio abanico de sistemas es necesario disponer de diferentes métodos de análisis y diseño. Los circuitos secuenciales tienen dos usos principales en los computadores:

a) Como *elementos de memoria*. Almacenan la información mientras está siendo procesada.

b) Como *circuitos de control*. Generan las señales de control necesarias para seleccionar y habilitar una secuencia de pasos en la transferencia o en el procesamiento de datos.

Ambos circuitos, combinacionales y secuenciales, operando conjuntamente proporcionan todas las funciones de almacenamiento, transferencia, procesamiento y control necesarias en un sistema digital como es el computador.

En este apéndice, en primer lugar se formaliza la noción de máquina secuencial y se introducen las ideas básicas que permiten modelar comportamientos con esta naturaleza. El concepto

fundamental es el de *estado* de una máquina secuencial, que *es toda la información que se necesita en cualquier instante de tiempo para poder conocer, dada la entrada en ese momento, cual será el valor de la salida.*

Metodología

Se ha tratado de cuidar de manera muy especial los aspectos específicos de la enseñanza a distancia. Los conceptos se introducen de forma progresiva, de manera que el estudiante pueda ir avanzando a su propio ritmo. La gran cantidad de figuras y tablas que contiene el texto (más de 600) tienen como objetivo facilitar su estudio sin la ayuda directa de un profesor.

La estructuración de todos los temas es uniforme. Todos contienen, además de las secciones específicas de desarrollo del tema, una sección de introducción en la que se exponen los objetivos a cubrir, una sección final de conclusiones en la que se resumen los conceptos introducidos en el tema y una sección de problemas con diferentes grados de dificultad.

Agradecimientos

Los autores expresan su agradecimiento más sincero a sus compañeros en los Departamentos de Informática y Automática e Inteligencia Artificial de la UNED en los que vienen desarrollado en los úlimos años su actividad universitaria por su apoyo en todo momento.

Muy particularmente este agradecimiento queremos personalizarlo en los profesores del Departamento de Informática y Automática, Roberto Hernández Berlinches, Juan Carlos Lázaro Obensa, Sebastián Dormido Canto y Raquel Dormido Canto por la lectura crítica y cuidadosa que han hecho de este texto. Sus sugerencias y atinados comentarios han contribuido a mejorar notablemente muchos de los conceptos expuestos. El diseño de la portada del texto ha sido realizada por Juan Carlos Lázaro Obensa.

Han aportado muchas ideas al texto original los profesores Ángel Pérez de Madrid Pablo y Pablo Ruipérez García que imparten la asignatura de Estructura y Tecnología de Computadores II a los alumnos de la Escuela de Informática de la UNED. También deseamos expresar nuestro agradecimiento a los alumnos de la Escuela de Ingeniería Técnica Informática de la UNED Oscar Goetsch, Carlos Maltese Girón y Manuel Montealegre Sánchez que han dasarrollado el simulador de la unidad de control microprogramable ucμP del computador SIMPLE2 bajo la supervisión del profesor Angel Pérez de Madrid Pablo.

Los autores

Tema 1

Estructuras de interconexión de un computador

En su nivel jerárquico superior, un computador consta de la *unidad central de proceso* (*CPU*[1]), la *memoria* y la *unidad de E/S* (uno o más módulos de cada tipo). Estos componentes se interconectan de una forma determinada para ejecutar programas que es la función básica de todo computador. Así pues, se puede describir un computador si se conoce lo siguiente:

1) La conducta externa de cada componente, esto es, los datos y señales de control que intercambia con otros módulos.

2) La estructura de interconexión y las órdenes de control que se necesitan para gestionar su utilización.

Esta visión macroscópica de la estructura y funcionamiento de un computador resulta particularmente importante por su capacidad para explicar y entender la naturaleza del mismo.

El conocimiento adquirido a este nivel del funcionamiento de un computador, permite tener una mejor comprensión de cuales son los puntos débiles de las diferentes alternativas, la naturaleza de un mal funcionamiento del sistema global si falla un componente y la facilidad que se tiene para mejorar sus prestaciones. En muchos casos la consecución de alguno de estos requisitos se obtiene modificando el diseño global, en lugar de aumentando la velocidad y fiabilidad de los componentes individuales.

Como introducción al tema se hace una breve revisión de los componentes fundamentales de la arquitectura clásica de von Neumann. A continuación se analizan las estructuras básicas utilizadas para la interconexión de los elementos de un computador. El tema finaliza con el estudio del tipo de interconexión más usual en un computador: *la estructura de buses*. Un bus no es nada más que un camino de comunicación compartido, consistente en una o varias líneas. En algunos computadores, la CPU, la memoria y las unidades de E/S se conectan a un bus único. Como las líneas son compartidas por varios componentes sólo uno de ellos puede transmitir a la vez. Se describen en este tema la estructura general de un bus, cuáles son sus parámetros de diseño más relevantes y ciertas consideraciones prácticas en la conexión mediante buses.

---

1. A lo largo de todo el texto se utilizará el término *CPU* que es el acrónimo de (*Central Process Unit*) para designar a la unidad central de proceso.

Sección 1-1

Componentes de un computador

La mayoría de los computadores actuales se basan en las ideas que el matemático, de origen húngaro y nacionalizado norteamericano, John von Neumann desarrolló en el Institute for Advanced Studies de Princeton. Lo que hoy día se conoce como *arquitectura de von Neumann* se fundamenta en tres ideas claves:

a) En la memoria del computador se almacenan indistintamente datos e instrucciones.

b) Se puede acceder a la información contenida en la memoria especificando la dirección donde se encuentra almacenada. La memoria no distingue si lo que contiene es un dato o una instrucción.

c) La ejecución de un programa, entendido como un conjunto de instrucciones, se realiza de forma secuencial pasando de una instrucción a la que le sigue inmediatamente, a no ser que explícitamente se modifique esta secuencia mediante una ruptura, ya sea condicional o incondicional.

Existe un pequeño conjunto de componentes lógicos básicos que se pueden combinar de diferentes formas para almacenar datos binarios y realizar operaciones aritmético-lógicas sobre ellos. Si se desea efectuar un determinado cálculo es posible diseñar una configuración "ad hoc" utilizando los componentes que resuelven el problema planteado.

En cierta manera se puede ver este procedimiento de interconexión de componentes como una forma de programación. El programa resultante está implícito en la propia estructura física (hardware) y desde este punto de vista se habla de un programa cableado. Este enfoque de la programación es, obviamente, muy poco flexible ya que resuelve exclusivamente el problema específico para el que fue diseñado y modificarlo resulta lento y costoso.

Otra alternativa consiste en considerar una configuración de módulos aritméticos y lógicos de propósito general. Este conjunto de elementos efectúa distintas funciones sobre los datos, dependiendo de las señales de control que en cada momento se les aplica.

En el primer caso se ha diseñado un sistema a medida que acepta unos datos de entrada y produce unos resultados de salida (ver Figura 1.1). En el segundo caso el sistema es de propósito general y además de los datos para producir los resultados acepta señales de control. Por este motivo, en lugar de modificar el conjunto de funciones lógicas y aritméticas de que se dispone para cada nuevo programa, únicamente se cambia el conjunto de señales de control.

La pregunta sería ¿cómo se generan las señales de control? Se debe tener presente que el programa completo es una secuencia de pasos o acciones elementales. En cada paso se realiza alguna operación aritmético-lógica sobre los datos y por lo tanto se necesitará un nuevo conjunto de señales de control, dependientes de la acción que se realice. La solución consiste en proporcionar un único código para cada po-

sible conjunto de señales de control, e incorporar un nuevo módulo o unidad en el sistema de cálculo que pueda aceptar los diferentes códigos de instrucciones y generar las señales de control adecuadas (ver Figura 1.2).

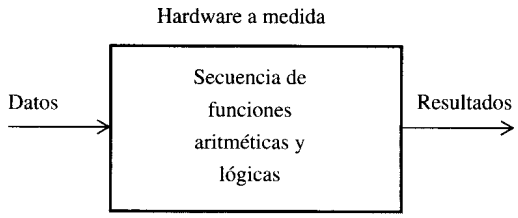


Figura 1.1: Programación por hardware (Programa cableado)

En resumen, la programación es ahora una tarea mucho más flexible. En lugar de volver a diseñar (modificar) la interconexión de los componentes lógicos que resuelven el problema cuando éste cambia, todo lo que hay que hacer es proporcionar una nueva secuencia de códigos. Cada código es una instrucción, y una unidad del sistema la interpreta (traduce) y genera las señales de control adecuadas.

La Figura 1.2 muestra los dos componentes básicos del sistema:

1) Un intérprete de instrucciones que se denomina **unidad de control**.
2) Un módulo de funciones lógicas y aritméticas de propósito general que se conoce como **unidad aritmético-lógica** (ALU[2]).

Estas dos unidades conjuntamente constituyen la **unidad central de proceso** (CPU).

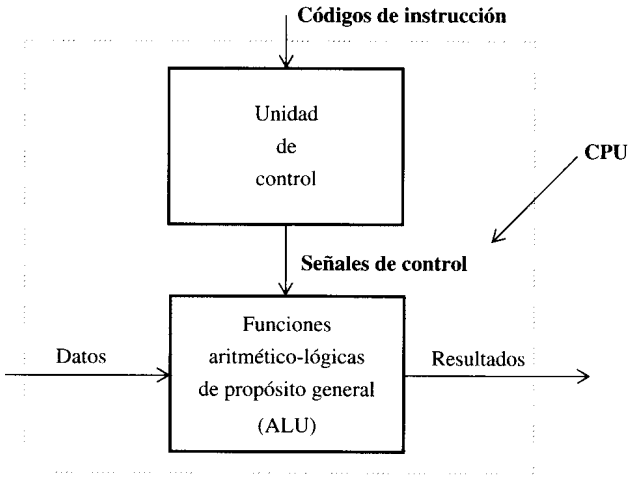


Figura 1.2: Programación por software

2. A lo largo de todo el texto se utilizará el término *ALU* que es el acrónimo de (*Arithmetic Logical Unit*) para designar a la unidad aritmético-lógica.

Para que el computador pueda funcionar necesita algunos componentes adicionales. Hay que introducir en el sistema tanto los datos como las instrucciones y para ello se utiliza una unidad de entrada. Este módulo contiene los componentes básicos que le permite aceptar datos e instrucciones y convertirlos a un formato interno capaz de ser utilizado por el sistema. También se precisa una unidad de salida para poder sacar los resultados al mundo exterior. Ambas unidades se conocen como **unidad de entrada/salida** (E/S).

La unidad de entrada introduce las instrucciones y los datos de forma secuencial, sin embargo, un programa no se ejecuta invariablemente siguiendo una estructura lineal, ya que en ella pueden existir rupturas (incondicionales o condicionales). De manera análoga, hay determinadas operaciones a realizar sobre los datos que pueden necesitar acceder en un instante de tiempo y en una secuencia predeterminada a más de un elemento. Esto exige que exista un lugar para almacenar, de forma temporal, tanto los datos como las instrucciones. El módulo o unidad encargado de esta función se conoce como **unidad de memoria o memoria principal**, para distinguirlo de la memoria externa o auxiliar.

Una de las ideas revolucionarias de von Neumann fue precisamente el concepto de programa almacenado en memoria, puesto que la misma unidad de memoria se puede utilizar para almacenar instrucciones y datos. *El que el contenido de una dirección de memoria sea considerado como un dato o como una instrucción depende únicamente de quien recibe esa información.* Si la acepta la ALU se la considera un dato sobre el que se efectúa un determinado cálculo, y si quien lo toma es la UC será interpretado como un código de instrucción para generar las correspondientes señales de control. En la Figura 1.3 se representan estos componentes.

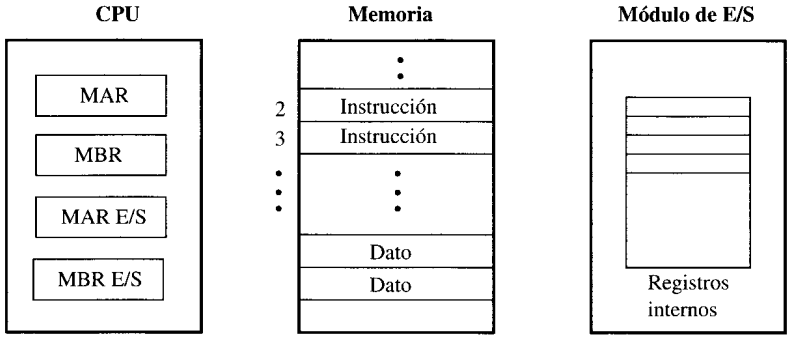


Figura 1.3: Componentes de un computador: Visión macroscópica

En su modo de funcionamiento normal, es la CPU quien tiene el control del computador y opera intercambiando datos con la memoria. Para hacer esto la CPU dispone de dos registros internos:

1) **Registro de dirección de memoria** (*MAR*[3]). Contiene la próxima dirección de memoria de donde se va a leer o donde se va a escribir.

2) **Registro de datos de memoria** (*MBR*[4]). Contiene el dato a escribir en la memoria o recibe el dato leído de la memoria.

3. A lo largo del texto *MAR* es el acrónimo de (*Memory Address Register*) y designa al registro de dirección de memoria.
4. A lo largo del texto *MBR* es el acrónimo de (*Memory Buffer Register*) y designa al registro de datos de memoria

El módulo de E/S también dispone de registros internos análogos:

1) **Registro de dirección de E/S** (*MAR E/S*). Especifica un dispositivo particular de E/S.
2) **Registro de datos de E/S** (*MBR E/S*). Se utiliza para el intercambio de datos entre un módulo de E/S y el resto del sistema.

La unidad de memoria consiste en un conjunto de posiciones definidas por sus direcciones que están dadas de forma secuencial. Cada posición contiene un número binario que se puede interpretar como un dato o como una instrucción. Un módulo de E/S transfiere datos desde los dispositivos externos a la CPU y a la memoria y viceversa. Para realizar esta función dispone de registros internos que almacenan temporalmente estos datos hasta que puedan ser enviados o recibidos.

Una vez examinados de forma muy esquemática, y a modo de recordatorio, las unidades básicas que constituyen un computador, se analizará como operan conjuntamente estas componentes para ejecutar programas.

Sección 1-2

Función de un computador

La función básica que realiza un computador es la ejecución de un programa. Un programa consiste en un conjunto de instrucciones y datos almacenadas en la unidad de memoria. La CPU es la encargada de ejecutar las instrucciones especificadas en el programa. Con el fin de obtener una mejor comprensión de esta función y de la forma en que las componentes principales del computador interaccionan para ejecutar un programa, se necesita analizar con más detalle el proceso de ejecución.

La secuencia de operaciones realizadas en la ejecución de una instrucción constituye lo que se denomina *ciclo de instrucción*. El punto de vista más sencillo es considerar que el procesamiento del ciclo de instrucción consta de dos pasos:

a) Fase o ciclo de búsqueda

b) Fase o ciclo de ejecución

Durante el ciclo de búsqueda (que es una operación común al conjunto de todas las instrucciones) la instrucción se localiza en la memoria y se lee. El ciclo de ejecución aunque depende, obviamente, de la naturaleza de la instrucción que se va a ejecutar, normalmente requiere la decodificación de la instrucción, la búsqueda de los operandos y la ejecución de la operación especificada por el código de la instrucción. La ejecución de un programa consiste en la repetición cíclica de las fases de búsqueda y ejecución de las instrucciones de que consta y finaliza mediante una de las siguientes acciones:

1) si se desconecta el computador

2) si ocurre algún tipo de error irrecuperable

3) si se ejecuta en el propio programa una instrucción que fuerza su terminación.

En la Figura 1.4 se representa la descripción de las dos fases que constituyen el ciclo de instrucción.

1.2.1 Ciclos de búsqueda y ejecución

En el comienzo de cada ciclo de instrucción la CPU busca en la memoria una instrucción. En una CPU tipo von Neumann para realizar esta tarea se dispone de un registro especial llamado **contador de programa** (*PC*[5]), que se utiliza para llevar el control de la posición de memoria donde tiene que ir a buscar la próxima instrucción. A menos que se indique lo contrario, la CPU siempre incrementa el PC después de cada fase de búsqueda de la instrucción. De esta manera irá a buscar la próxima instrucción de forma secuencial (es decir

5. A lo largo del texto *PC* es el acrónimo de (*Program Counter*) y designa al registro contador de programa.

la instrucción que se encuentra localizada en la dirección de memoria siguiente).

Sea, por ejemplo, un computador en el que cada instrucción ocupa una palabra de memoria de 16 bits. Se supone que el contador de programa apunta a la posición 500 de la memoria y será en esta posición donde la CPU buscará la próxima instrucción. En los sucesivos ciclos de instrucción leerá las siguientes instrucciones de las posiciones 501, 502, 503 etc. No obstante, esta secuencia, como se verá más adelante, se puede alterar por la ejecución del propio programa.

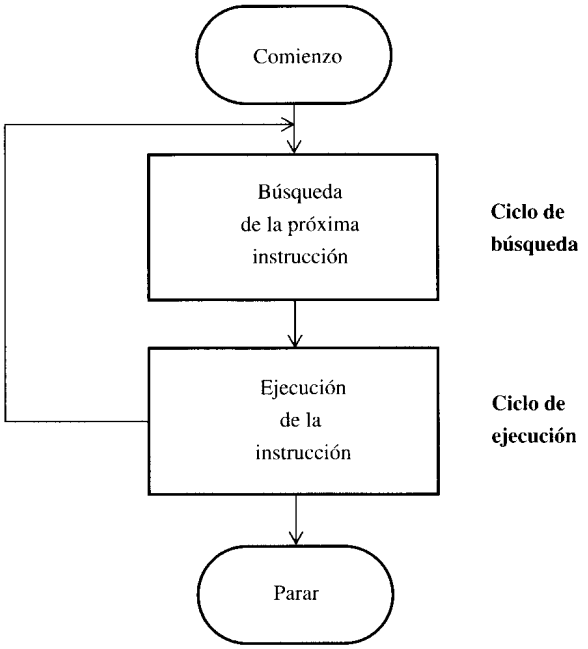


Figura 1.4: Ciclo básico de una instrucción

La instrucción leída se carga (almacena) en un registro de la CPU conocido como **registro de instrucción** (*IR*[6]). La instrucción, que está codificada en binario, especifica el tipo de acción que tiene que llevar a cabo la CPU. Ésta interpreta la instrucción y ejecuta la acción indicada. En general estas acciones pertenecen a una de las cuatro categorías siguientes:

- *CPU-Memoria*: Los datos se pueden transferir desde la CPU a la memoria o desde la memoria a la CPU.
- *CPU-E/S*: Los datos se pueden transferir a o desde el mundo exterior mediante el intercambio entre la CPU y un módulo de E/S.
- *Procesamiento de los datos*: La CPU puede realizar un conjunto básico de operaciones aritméticas o lógicas sobre los datos.

6. A lo largo del texto *IR* es el acrónimo de (*Instruction Register*) y designa al registro de instrucción.

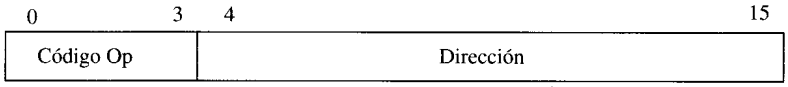
- *Control*: Una instrucción puede especificar que se altere la secuencia de ejecución. Por ejemplo, la CPU lee la instrucción de la posición 234, que expresa que la siguiente instrucción se leerá de la posición 315. La CPU recordará este hecho fijando el PC a 315, por lo que en el siguiente ciclo de búsqueda se leerá la instrucción almacenada en la posición 315, en lugar de la que hay en la posición 235.

La ejecución de una instrucción puede necesitar una combinación de estas acciones.

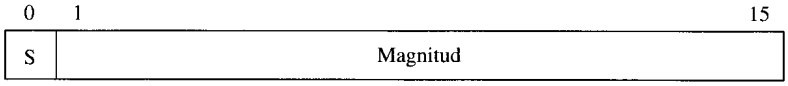
1.2.2 Ejemplo: Ejecución de una instrucción

Se considera una máquina hipotética que incluye las características especificadas en la Figura 1.5.

a) Formato de instrucción:



b) Formato de los números enteros:



c) Registros internos de la CPU:

Contador de Programa (PC) = Dirección de la próxima instrucción

Registro de Instrucción (IR) = Instrucción que se está ejecutando

Acumulador (AC) = Almacenamiento temporal

d) Lista parcial de códigos de operación:

0001 = Cargar el AC desde memoria

0010 = Almacenar el AC en memoria

0110 = Restar al AC de memoria

Figura 1.5: Características de un computador hipotético

La CPU contiene un acumulador (AC) que permite almacenar los datos de forma temporal. Tanto las instrucciones como los datos tienen una longitud de 16 bits. El formato de la instrucción indica que puede haber $2^4 = 16$ códigos de operación diferentes y hasta $2^{12} = 4.096$ (4K) palabras de memoria con capacidad de direccionamiento directo, ya que en el formato de instrucción se han reservado 4 bits para el código de operación y 12 bits para la dirección.

La Figura 1.6 ilustra la ejecución parcial de un programa, mostrando las partes relevantes de la memoria

y de los registros de la CPU. Se ha utilizado, como es habitual, una notación hexadecimal. El fragmento de programa que se muestra resta al contenido de la posición de memoria 750_{16} el contenido de la posición de memoria 751_{16} y almacena el resultado en la posición de memoria 751_{16}. Para realizar esta acción son necesarias tres instrucciones que se pueden describir con tres ciclos de búsqueda y tres ciclos de ejecución de la forma siguiente:

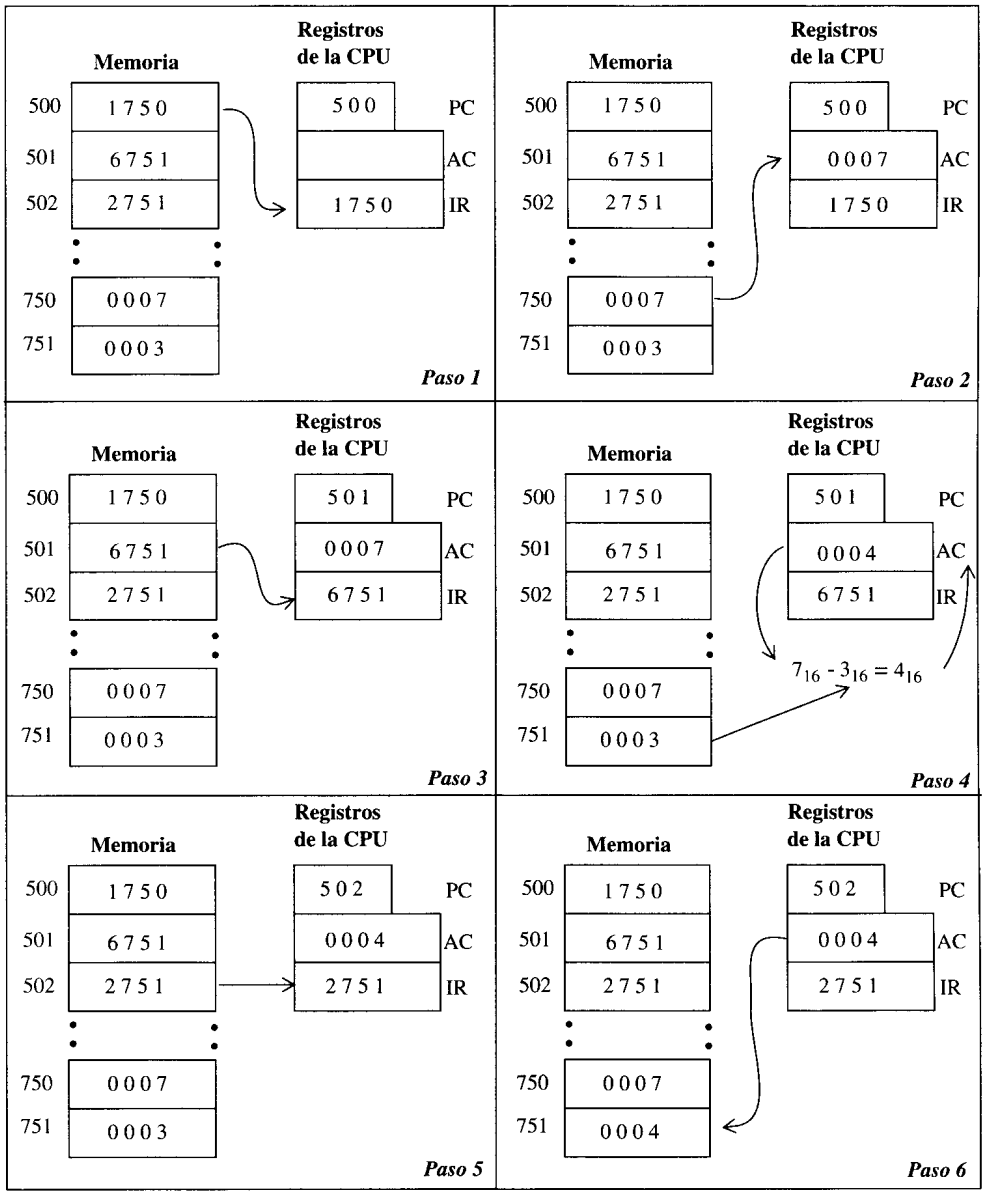


Figura 1.6: Ejemplo de ejecución de un programa

1) El contenido del contador de programa (PC) es 500_{16}, que es la dirección de la primera instrucción. El contenido de esta dirección se carga en el registro de instrucción (IR). Debe observarse que este proceso requiere la utilización del registro de dirección de memoria (MAR) y un registro de datos de memoria (MBR). Para hacer el ejemplo de ejecución del programa lo más sencillo posible se ignoran, por el momento, los contenidos de estos registros internos a la CPU.
2) Los 4 primeros bits de IR (0001) indican que hay que cargar el AC. Los 12 bits restantes especifican la dirección, que es 750_{16}.
3) Se incrementa el PC y se lee la instrucción siguiente.
4) Al antiguo contenido del AC, se le resta el contenido de la posición 751_{16} y el resultado se almacena en el acumulador (código de operación = 0110).
5) Se incrementa el PC y se lee la instrucción siguiente.
6) Se almacena el contenido del AC en la posición 751_{16} (código de operación = 0010).

En este ejemplo, para restar al contenido de la posición 750_{16} el contenido de la posición 751_{16} se necesitan tres ciclos de instrucción, cada uno consistente en un ciclo de búsqueda y un ciclo de ejecución. Con un conjunto de instrucciones más complejas se habrían necesitado menos ciclos. ♦

1.2.3 Diagrama de flujo del ciclo de instrucción

La mayoría de las CPU's modernas incluyen instrucciones que requieren más de una posición de memoria. La instrucción *SUB B, A* del computador PDP-11, almacena la resta de los contenidos de las posiciones de memoria *B* y *A* en la posición de memoria *A*. De este modo la operación completa se efectúa en un único ciclo de instrucción con los siguientes pasos:

1) Leer la instrucción *SUB*.
2) Leer el contenido de la posición de memoria *A* en la CPU.
3) Leer el contenido de la posición de memoria *B* en la CPU. Para que no se pierda el contenido de *A*, la CPU debe tener al menos dos registros internos que permitan almacenar los datos traídos de la memoria.
4) Restar los dos valores.
5) Escribir el resultado generado en la CPU en la posición de memoria *A*.

Se ve así que el ciclo de ejecución de una instrucción particular puede necesitar más de una referencia a memoria. Por otro lado una instrucción también puede especificar una operación de E/S. Teniendo en cuenta estas consideraciones, en la Figura 1.7 se presenta un análisis del ciclo de instrucción básico más detallado que el de la Figura 1.4. Se representa en la forma de un diagrama de flujo. Para cualquier ciclo de instrucción dado, algunos estados pueden ser nulos y otros visitarse más de una vez. El significado de los diferentes estados es el siguiente:

- *Cálculo de la dirección de la instrucción (CDI)*: Determina la dirección de la próxima instrucción que se tiene que ejecutar. Normalmente esto se hace sumando un número fijo a la dirección de la instrucción anterior. Por ejemplo, si cada instrucción tiene una longitud de 16 bits y la memoria

está organizada en palabras de 16 bits, se suma 1 a la dirección previa. Si la memoria se estructura en bytes (8 bits de longitud) se suma 2 para calcular la nueva dirección.

- *Búsqueda de la instrucción (BI)*: Leer la instrucción de su posición de memoria en la CPU.
- *Decodificación de la instrucción (DI)*: Analizar la instrucción para determinar el tipo de operación que se va a efectuar y los operandos que se utilizan.

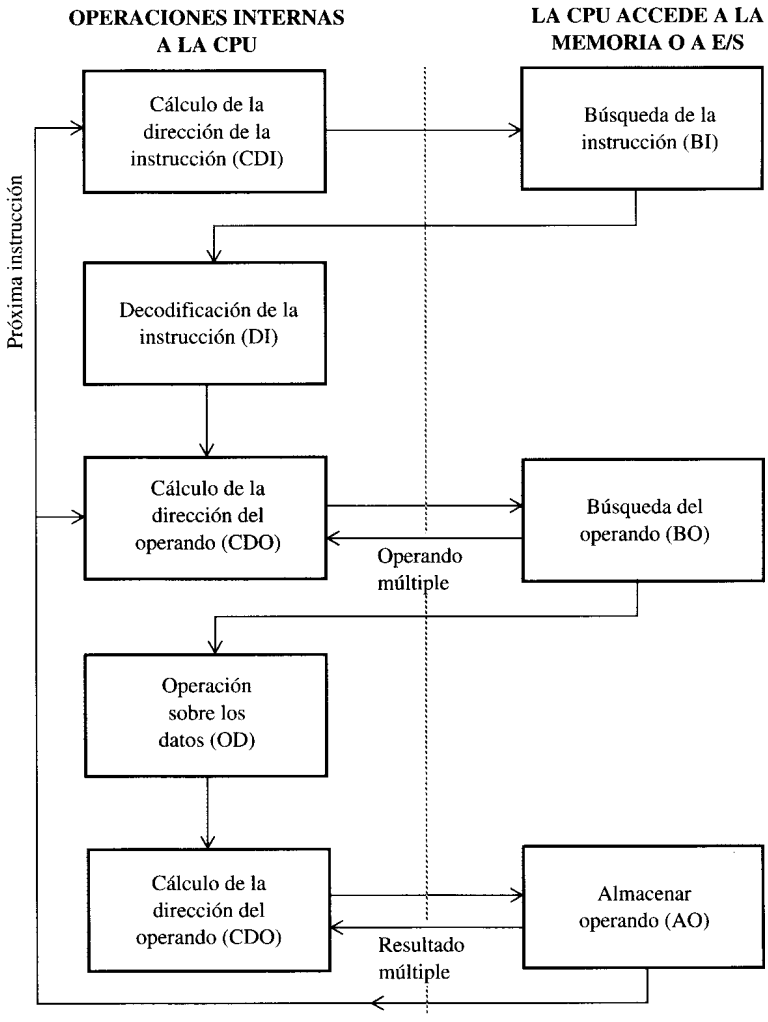


Figura 1.7: Diagrama de estado del ciclo de instrucción

- *Cálculo de la dirección del operando (CDO)*: Si la operación realiza una referencia a un operando almacenado en la memoria o que está disponible a través de una operación de E/S, se determina la dirección del operando.
- *Búsqueda del operando (BO)*: Leer el operando de la memoria o de un dispositivo de E/S.

- *Operación sobre los datos (OD)*: Realizar la operación indicada en la instrucción.
- *Almacenar el operando (AO)*: Escribir el resultado en memoria o en un dispositivo de E/S.

Los estados que aparecen en la parte derecha de la Figura 1.7 efectúan un intercambio entre la CPU y la memoria o un módulo de E/S. Los estados de la parte izquierda del diagrama sólo hacen operaciones internas en la CPU. El estado CDO aparece dos veces ya que una instrucción puede necesitar una lectura, una escritura o ambas. Sin embargo, la acción que se lleva a cabo durante ese estado es básicamente la misma en ambos casos, de forma que sólo se necesita un único identificador de estado.

También se puede observar que el diagrama permite tanto operandos como resultados múltiples, debido a que algunas instrucciones en determinados computadores así lo necesitan. Por ejemplo, la instrucción *ADD A, B* en el computador PDP-11 produce la siguiente secuencia de estados: *CDI, BI, DI, CDO, BO, CDO, BO, OD, CDO, AO*. En algunas máquinas una única instrucción puede especificar la operación a realizar sobre un vector de números o una cadena de caracteres (array unidimensional). Tal como se muestra en la Figura 1.7 esto exigiría repetidas búsquedas de operandos y/u operaciones de almacenamiento.

1.2.4 El ciclo de interrupción

Potencialmente, todos los computadores disponen de un mecanismo mediante el cual otros módulos (E/S, memoria) pueden interrumpir el procesamiento normal de la CPU. La naturaleza específica de estas interrupciones se verá, con cierto detalle, en el Tema 3 que trata de la unidad de entrada/salida. No obstante, se necesita introducir en este punto el concepto de interrupción para poder comprender más fácilmente la naturaleza del ciclo de instrucción y las implicaciones de las interrupciones sobre la estructura de interconexión. Ahora, el interés no se centra en los detalles de la generación y procesamiento de las interrupciones sino en la comunicación entre módulos que se produce como consecuencia de ellas.

El objetivo fundamental de las interrupciones es mejorar el rendimiento de los computadores desde el punto de vista de su velocidad de procesamiento. Como la mayoría de los dispositivos periféricos son mucho más lentos que la CPU, si se supone que ésta está transfiriendo datos a una impresora utilizando el esquema del ciclo de instrucción de la Figura 1.4, después de cada operación de escritura la CPU tendrá que parar y permanecer en un estado de espera hasta que la impresora finalice su tarea. El tiempo de esta pausa puede ser del orden de algunos cientos o incluso miles de ciclos de instrucción en los que la CPU no interacciona con la memoria. Obviamente, esta situación corresponde a un mal uso de la CPU. Si se utilizasen interrupciones la CPU podría estar ocupada ejecutando otras instrucciones mientras se está realizando la operación de E/S.

Para aclarar el concepto se va a suponer que la CPU tiene dos programas: uno que lee datos de la memoria y los transfiere a un dispositivo periférico; el otro es una aplicación numérica que necesita una gran cantidad de cálculo. La CPU inicia el primer programa emitiendo una orden de escritura al dispositivo periférico y a continuación comienza la ejecución de la otra aplicación. Si el dispositivo de salida está ya preparado para continuar, esto es, cuando puede aceptar más datos de la CPU, el módulo de E/S para ese periférico envía a la CPU una señal de *petición de interrupción*. La CPU responde suspendiendo la operación del programa que tiene en ejecución y activando el programa que presta servicio a ese dispositivo particular de E/S. Al terminar de atender esa petición, la CPU vuelve otra vez al programa original y continúa su ejecución en el punto en el que lo interrumpió.

Para permitir este mecanismo de interrupciones se hace necesario añadir en el ciclo de instrucción, tal como se muestra en la Figura 1.8, un *ciclo de interrupción*. En este ciclo la CPU comprueba si se ha producido alguna interrupción y en caso afirmativo indica su presencia mediante una señal apropiada. Si no

existe ninguna señal de interrupción que esté pendiente de ser atendida, la CPU continúa con el ciclo de búsqueda de la siguiente instrucción del programa que tiene en ejecución. Si por el contrario la CPU observa que está activada la señal de interrupción (lo que indica que hay al menos una interrupción pendiente de ser atendida) hace lo siguiente:

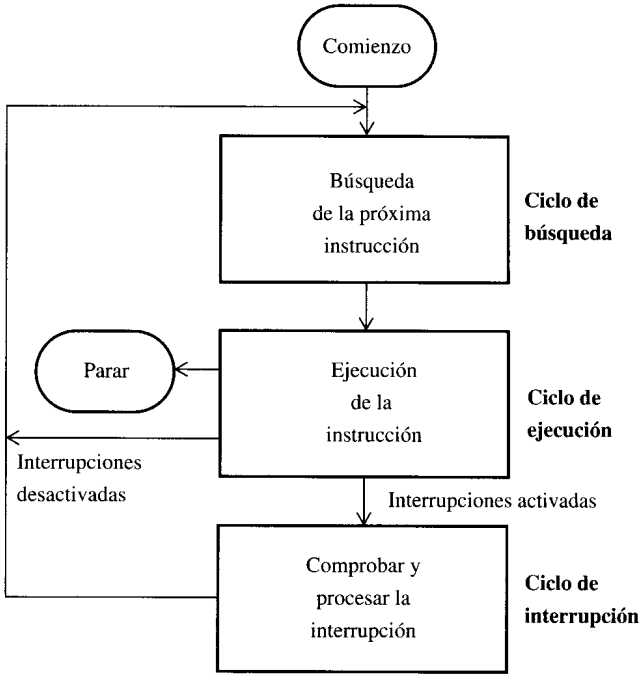


Figura 1.8: Ciclo de instrucción con interrupciones

1) Guarda el contexto del programa que en ese momento se está ejecutando. Esto significa almacenar la dirección de la próxima instrucción que se va a ejecutar (contenido actual del contador de programa) y cualquier otro dato que sea relevante a la actividad de la CPU, para poder continuar posteriormente la ejecución de este programa en el punto que lo interrumpió (por ejemplo el contenido de todos los registros internos de la CPU).

2) Carga en el contador de programa la dirección de comienzo de un programa encargado de *gestionar la interrupción.*

La CPU procede ahora con el ciclo de búsqueda. Para ello lee la primera instrucción del programa que gestiona la interrupción y que se encarga de dar servicio a la petición que estaba solicitada y pendiente. Todos los sistemas operativos incorporan un programa gestor de interrupciones. Normalmente este programa se ocupa de analizar la naturaleza de la interrupción y de realizar cualquier acción que se necesite. Así, en el ejemplo que se está utilizando, el gestor determina qué módulo de E/S generó la interrupción y se dirige a un programa específico que transfiere más datos a dicho módulo de E/S. Como se guardó el contexto del programa que estaba activo al producirse la interrupción, la CPU continua con su ejecución una vez deja preparado al programa que gestiona la interrupción (que se encarga de enviar los datos al periférico de salida).

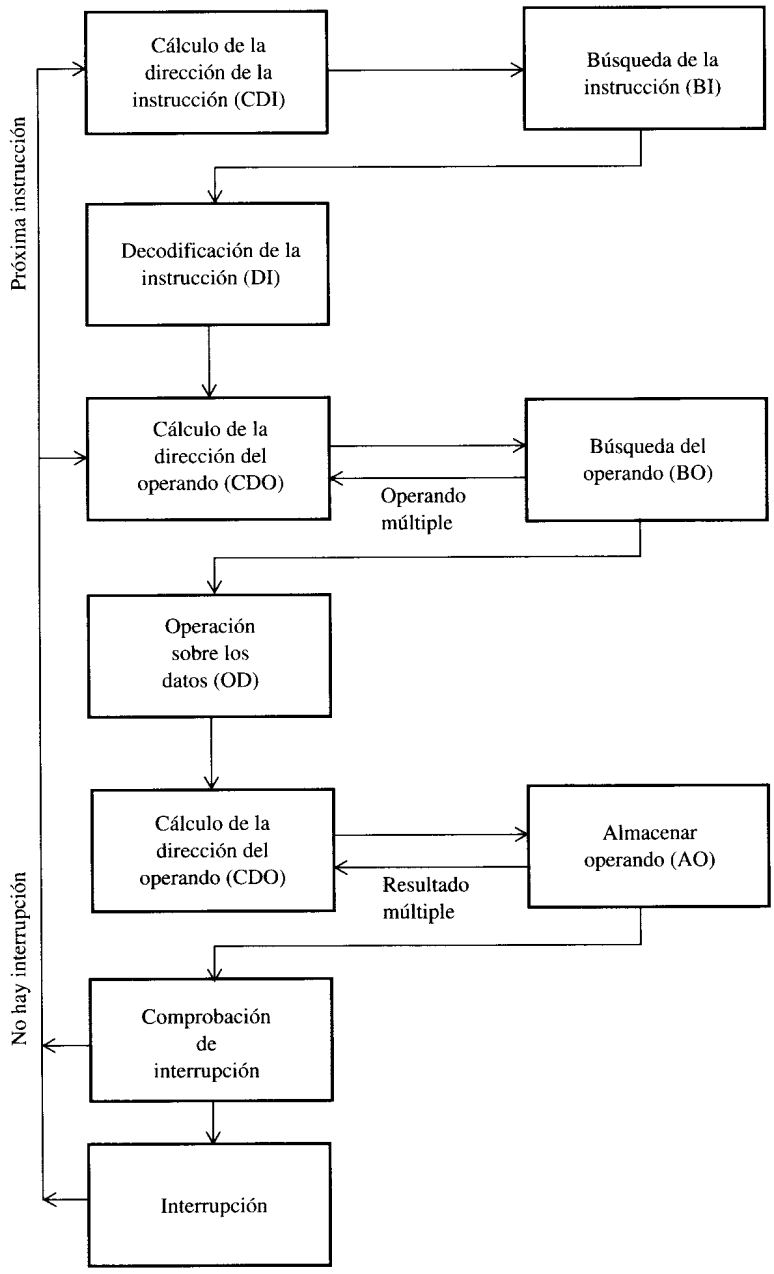


Figura 1.9: Diagrama de flujo del ciclo de instrucción con interrupciones

Es evidente que en todo este proceso se incurre en un cierto retardo, ya que deben ejecutarse instrucciones extras (en el gestor de interrupciones) para determinar la naturaleza de la interrupción y decidir cual es la acción apropiada. Sin embargo, como el tiempo que se pierde si se tiene que estar esperando a que

finalice una operación de E/S es relativamente grande, el mecanismo de interrupciones permite un uso mucho más eficaz de la CPU.

Las interrupciones no se atienden siempre de forma instantánea. Es posible desactivarlas todas o de forma selectiva y posteriormente volverlas a activar. Una *interrupción desactivada* significa que la CPU ignorará esa señal de petición de interrupción. Por ejemplo, suele ser deseable, generalmente, finalizar el procesamiento de una interrupción antes de atender a otra, por lo que el mecanismo de interrupciones a menudo se desactiva cuando la CPU está atendiendo a una de ellas. Si durante ese tiempo ocurre una nueva petición de interrupción quedará pendiente y será comprobada por la CPU una vez que haya vuelto a activar el mecanismo de interrupciones. En la Figura 1.9 se muestra el diagrama de flujo del ciclo de instrucción incluyendo el tratamiento de las interrupciones.

1.2.5 Función de E/S

Hasta ahora se ha visto el funcionamiento de un computador como un sistema que está controlado por la CPU y, básicamente, se ha analizado la interacción de esta unidad y la memoria, aludiéndose muy de pasada al papel de las unidades de E/S. Aunque esto se estudiará con cierto detalle en el tema 3, se da a continuación una breve introducción a estas unidades.

Un módulo de E/S puede intercambiar datos directamente con la CPU. De la misma forma que la CPU puede iniciar una operación de lectura o escritura con la memoria designando simplemente la dirección de una posición específica, la CPU también puede leer o escribir datos en un módulo de E/S. En este último caso, la CPU identifica a un dispositivo específico que está controlado por un módulo de E/S particular. Se puede plantear así, con instrucciones de E/S, una secuencia de acciones similar a la de la Figura 1.6 (que sólo contiene instrucciones con referencia a memoria).

A veces puede ser deseable que los intercambios de E/S ocurran directamente con memoria. En tal caso, la CPU cede al módulo de E/S la capacidad de leer o escribir en memoria, de manera que la transferencia E/S-memoria ocurre sin la intervención directa de la CPU. Durante una transferencia de este tipo, el módulo de E/S emite peticiones de lectura o escritura a la memoria, lo que libera a la CPU de la responsabilidad de ese intercambio. Esta operación se conoce como *acceso directo a memoria* (*DMA*[7]) que se estudiará en el tema 3. Por ahora todo lo que se precisa saber es que la estructura de interconexión del computador puede necesitar interacciones directas entre la memoria y los módulos de E/S.

7. A lo largo del texto *DMA* es el acrónimo de (*Direct Memory Access*) y designa el acceso directo a memoria.

Sección 1-3

Estructuras de interconexión

Un computador consiste en un conjunto de componentes o módulos de tres tipos básicos (CPU, memoria y E/S) que se comunican entre sí. Evidentemente deben existir caminos que conecten unos módulos con otros.

El conjunto de caminos que conectan los diferentes módulos es lo que constituye la *estructura de interconexión*. El diseño de esta estructura dependerá de los intercambios que se tengan que hacer entre los módulos. En la Figura 1.10 se muestran los tipos de intercambio que se necesitan, indicando las principales entradas y salidas para cada clase de módulo:

1) *Memoria*: Un módulo de memoria consta de N palabras de igual longitud. A cada palabra se le asigna una única dirección numérica (0, 1, .., N-1). Una palabra de datos se puede leer de (o escribir en) memoria. La naturaleza de la operación a efectuar se indica mediante dos señales de control de **Leer** y **Escribir**. La posición donde se realiza la operación se especifica por una dirección.

2) *Módulo de E/S*: Desde un punto de vista interno al computador, la unidad de E/S es funcionalmente similar a la memoria. Hay también dos operaciones de **Leer** y **Escribir**. Un módulo de E/S puede controlar más de un dispositivo externo. A cada una de las interfases del módulo de E/S con un dispositivo externo se le llama un *puerto*, que se referencia mediante una única dirección (0, 1, ..., M-1). Existen caminos de datos externos para la entrada y salida de información desde un dispositivo periférico. Un módulo de E/S puede enviar señales de interrupción a la CPU.

3) *CPU*: La CPU lee instrucciones y datos, escribe datos después de procesarlos y utiliza señales de control para dirigir la operación global del sistema. También recibe señales de interrupción.

Las entradas y salidas de los módulos de un computador, que se muestran en la Figura 1.10, definen los datos que se tienen que intercambiar. La estructura de interconexión debe soportar los siguientes tipos de transferencias:

- *Memoria a CPU*: La CPU lee una instrucción o un dato de la memoria.
- *CPU a Memoria*: La CPU escribe un dato en memoria.
- *E/S a CPU*: La CPU lee datos de un dispositivo de E/S a través de un módulo de E/S.
- *CPU a E/S*: La CPU envía datos al dispositivo de E/S.
- *E/S a Memoria*: Un módulo de E/S intercambia datos directamente con memoria sin que tenga que pasar a través de la CPU, utilizando el mecanismo de acceso directo a memoria (DMA).
- *Memoria a E/S*: La memoria intercambia datos directamente con un módulo de E/S sin que tenga que pasar a través de la CPU, utilizando el mecanismo de acceso directo a memoria (DMA).

A lo largo de los años se han propuesto diferentes estructuras de interconexión para la arquitectura de un computador. La mayoría de ellas pertenece a uno de los cuatro tipos siguientes:

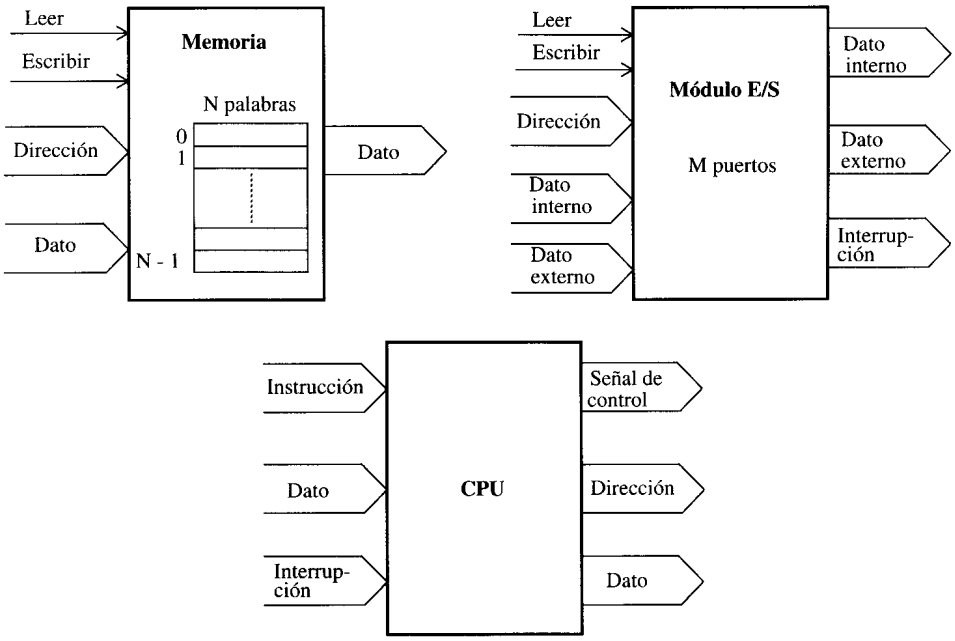


Figura 1.10: Módulos de un computador

- E/S a través de la CPU
- E/S a través de la memoria
- E/S mediante un conmutador central
- Bus de E/S

Cada una de estas arquitecturas se diseña para potenciar alguna ventaja en función de un entorno dado. Al mismo tiempo suelen introducir ciertas desventajas para otras funciones.

La *arquitectura de E/S a través de la CPU* (Figura 1.11) fue la adoptada inicialmente en los primeros computadores y es quizás la disposición más simple. En esta arquitectura la CPU y los módulos de E/S comparten la misma vía de acceso a memoria, lo que obliga a la CPU a parar momentáneamente sus cálculos cuando se están produciendo transferencias de datos. La CPU controla todos los intercambios y por lo tanto no es posible realizar un acceso directo a memoria desde los módulos de E/S. Todo el flujo de información entre la E/S y memoria debe pasar necesariamente a través de la CPU. Por su simplicidad, esta estructura de interconexión es la más económica desde un punto de vista hardware. Su principal desventaja es que se produce una reducción del rendimiento de la CPU debido a la actividad de los módulos de E/S.

En las primeras versiones de esta arquitectura, para cada operación que se realizaba entre la memoria y la unidad de E/S, la CPU era la encargada de leer del dispositivo de E/S y escribir en la memoria (o viceversa). En las versiones actuales se incorporan en la CPU circuitos adicionales que realizan las funciones

de gestión de la memoria. Incluso en este caso, la CPU y la E/S deben compartir estos recursos, lo que en último término afecta al rendimiento de la CPU.

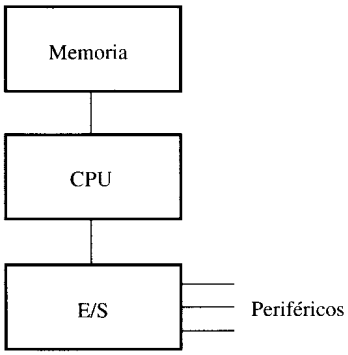


Figura 1.11: Arquitectura de E/S a través de la CPU

En la *arquitectura de E/S a través de la memoria* (Figura 1.12) es posible el acceso directo a la memoria principal de dos o más componentes de forma independiente.

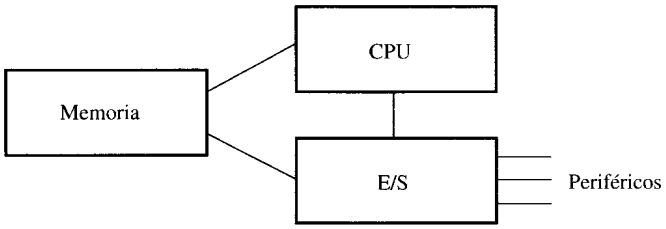


Figura 1.12: Arquitectura de E/S a través de la Memoria

El módulo de memoria contiene la lógica de control necesaria para resolver los conflictos. Una ventaja de este sistema es que puede realizarse directamente la transferencia entre la unidad de E/S y la memoria y mientras, simultáneamente, la CPU estar realizando otras tareas. Cuando la CPU desea iniciar una transferencia entre la unidad de E/S y la memoria, coloca una petición en una posición determinada de la memoria. El módulo de E/S comprueba periódicamente esa posición para ver si existe alguna petición. El punto débil de esta estructura es que los mecanismos de control son complejos y poco flexibles, es decir, es relativamente difícil añadir módulos adicionales.

En la *arquitectura de E/S mediante un conmutador central* (Figura 1.13) existe un mecanismo de distribución centralizado al cual se unen todos los componentes. El conmutador central controla el acceso a memoria tanto de la CPU como de los módulos de E/S. Como el camino de datos entre la E/S y la memoria no incluye a la CPU, ésta queda libre para realizar cálculos, de la misma forma que en la arquitectura E/S a través de la memoria. Al mismo tiempo puede suministrar a la CPU información de las tareas que se realizan y proporciona una conexión directa (que no pasa a través de la memoria) entre la CPU y la unidad de E/S. Debido a la potencia y flexibilidad de este enfoque, es la solución más adoptada en el caso de los grandes sistemas de computadores. Sus principales inconvenientes son el coste y la complejidad del conmutador.

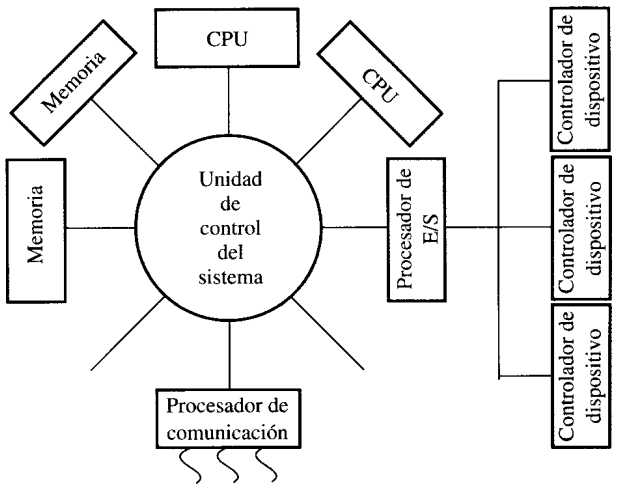


Figura 1.13: Arquitectura de E/S mediante un conmutador central

Una variante de la E/S a través de la memoria es la *arquitectura con bus de E/S* (Figura 1.14), que conecta los módulos de E/S con la memoria mediante unos controladores de dispositivos que son realmente "miniprocesadores de E/S". Esta arquitectura participa de muchas de las ventajas de la estructura anterior. En este caso hay un conjunto de líneas que se comparten por todos los módulos. En un instante de tiempo solamente un módulo puede transmitir de forma satisfactoria, y una o más de las unidades deben encargarse de controlar la utilización del bus. Por su simplicidad y flexibilidad, esta arquitectura se ha convertido en un estándar para los microcomputadores y minicomputadores. Dada su importancia se dedica el resto de este tema al estudio de las arquitecturas tipo bus.

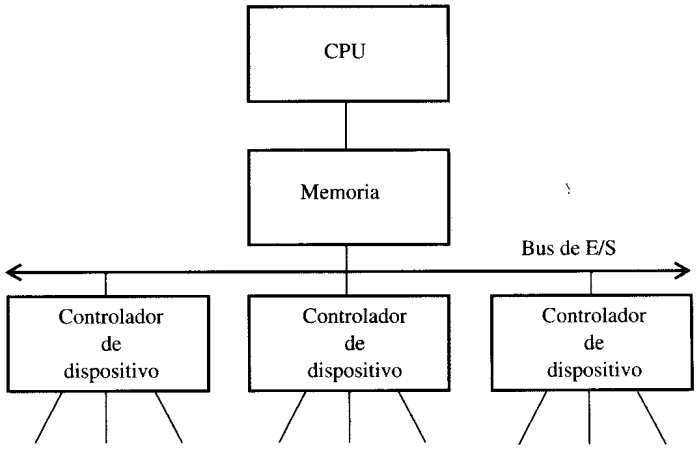


Figura 1.14: Arquitectura con bus de E/S

Sección 1-4

Interconexión mediante bus

Un bus es una vía de comunicación que conecta dos o más dispositivos (Figura 1.15). *La característica clave de un bus es que se trata de un medio de transmisión compartido.* Al bus se conectan múltiples dispositivos, y una señal transmitida por cualquiera de ellos puede ser recibida por todas las otras unidades conectadas. Si dos dispositivos intentasen transmitir durante el mismo período de tiempo sus señales se solaparían y no se podrían recuperar. Así pues, en un determinado instante de tiempo, solamente es posible la transmisión por parte de un único dispositivo. Sin embargo la recepción puede ser realizada por varios dispositivos simultánemente.

De estas consideraciones previas se deducen cuales deben ser las funciones de un bus:

1) Soportar la información a transmitir.

2) Garantizar la correcta comunicación entre los elementos que comparten el bus.

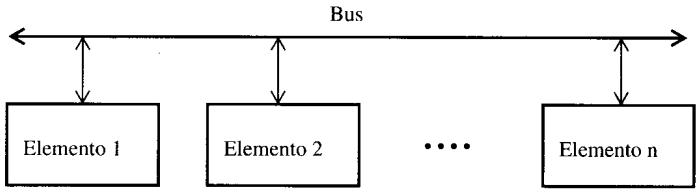


Figura 1.15: Arquitectura de bus

En muchos casos, un bus consiste en un conjunto de líneas de comunicación. Cada línea es capaz de transmitir señales que representan dos estados lógicos diferenciados (1 ó 0). Como en un intervalo de tiempo dado es posible transmitir una secuencia de dígitos binarios a través de una única línea, si se consideran de forma conjunta algunas líneas de un bus, se pueden usar para enviar dígitos binarios de forma simultánea (en paralelo). Por ejemplo, una unidad de información que conste de 8 bits se puede transmitir por un bus que tenga 8 líneas.

Los computadores suelen disponer de una serie de buses, que proporcionan vías de comunicación entre sus componentes en los diferentes niveles que constituyen su jerarquía. Se llama *bus del sistema* a aquel que conecta las unidades principales de un computador (CPU, memoria, E/S). Las estructuras de interconexión más frecuentes se basan en el uso de uno o más buses de este tipo.

1.4.1 Diagramas de temporización

En este apartado se van a introducir los diagramas de temporización para ilustrar como ocurren las secuencias de acciones y de relación de causa-efecto entre los diferentes sucesos.

La comunicación entre los dispositivos conectados a un bus tiene lugar a través de un conjunto de líneas capaces de llevar señales. Se pueden transmitir dos niveles diferentes de señal (niveles de tensión), que representan al 0 y al 1 binario respectivamente. Un diagrama de temporización muestra sobre una línea el nivel de la señal en función del tiempo (Figura 1.16).

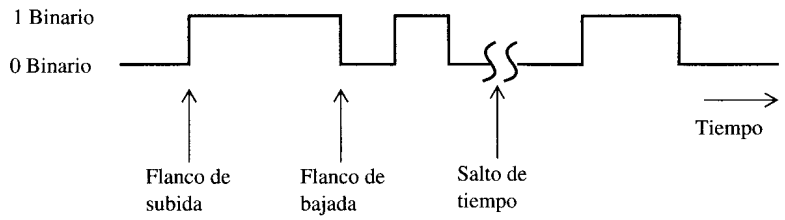


Figura 1.16: Señales binarias en función del tiempo

Por convenio se representa el 1 lógico en un nivel superior al del 0 lógico (lógica positiva), y normalmente este último nivel indica el valor por defecto de la señal (es decir, que si no se está transmitiendo ninguna señal por la línea, su nivel será el 0).

Se denomina *flanco de subida* a una transición de la señal de 0 a 1 y *flanco de bajada* cuando dicha transición es de 1 a 0. Por simplicidad en los diagramas se supone, aunque en la realidad esto no es así, que las transiciones ocurren de forma instantánea. De hecho, todas las transiciones requieren algo de tiempo para efectuarse, sin embargo, este tiempo suele ser muy pequeño si se le compara con la duración del nivel de la señal. A veces sucede que el valor de una variable es irrelevante durante determinados intervalos de tiempo. Esta circunstancia se representa mediante un salto en el eje temporal.

Con frecuencia las señales se representan formando grupos (Figura 1.17). Por ejemplo, si el dato que se transfiere en un instante de tiempo es un byte se necesitarán 8 líneas. Generalmente, no es importante conocer el valor exacto que se transmite por cada línea del grupo sino simplemente saber si hay o no señales presentes.

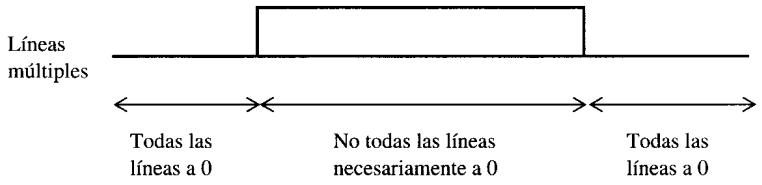


Figura 1.17: Grupo de líneas

La transición de una señal en una línea puede dar lugar a que se produzcan transiciones en las líneas de otros dispositivos conectados al bus. Así, si un módulo de memoria detecta una señal de control de lectura (por ejemplo transición de 0 a 1), colocará el dato correspondiente sobre las líneas de datos.

Esta relación de causa-efecto produce una secuencia de sucesos, y en el diagrama de temporización se utilizan pequeñas líneas con flechas para mostrar estas dependencias (Figura 1.18).

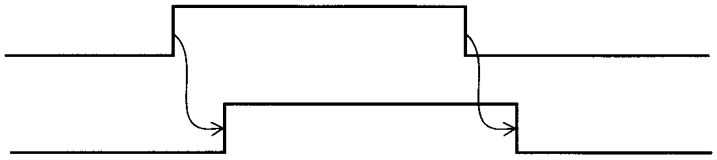


Figura 1.18: Dependencias de causa-efecto

Una línea de reloj suele formar parte, casi siempre, del bus del sistema. Se conecta un reloj electrónico a la línea de reloj que proporciona una secuencia, repetitiva y regular, de transiciones (ver Figura 1.19). Esta señal de reloj suele utilizarse para sincronizar otros sucesos.

Figura 1.19: Señal de reloj

1.4.2 Estructura de bus

Un bus del sistema consta de alrededor de 50 a 100 líneas independientes a las que se les asigna un significado o función particular. Aunque existe una gran variedad de diseño de buses en cualquiera de ellos las líneas se pueden clasificar en tres grupos funcionales (Figura 1.20):

- Líneas de datos
- Líneas de dirección
- Líneas de control

Además, existen las líneas de alimentación que suministran energía a los diferentes módulos y las líneas de referencia respecto de las cuales se fijan las tensiones de todas las demás (nivel de tierra).

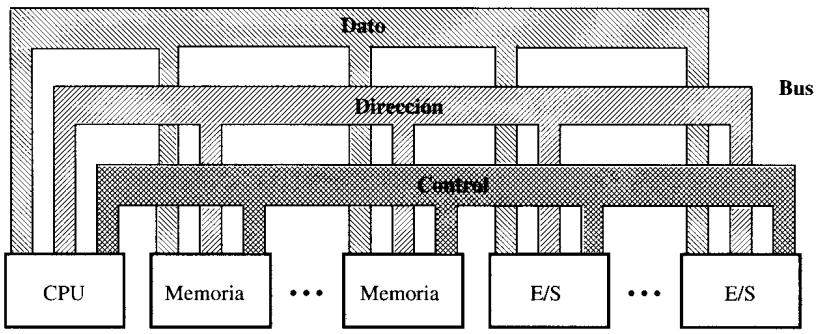


Figura 1.20: Esquema de interconexión de un bus

Las *líneas de datos* establecen un camino para transferir datos entre los módulos del sistema. El conjunto de estas líneas forman el *bus de datos*, que consta normalmente de 8, 16, 32, 64 o 128 líneas independientes (el número de líneas se conoce como la anchura del bus). Como en un instante de tiempo dado cada línea sólo puede transmitir un bit, el número de líneas determina cuantos bits se pueden transmitir de forma simultánea.

La anchura del bus de datos es un factor esencial para determinar el rendimiento global del sistema. Por ejemplo, si el bus de datos tiene una anchura de 8 bits y cada instrucción tiene una longitud de 16 bits, la CPU debe acceder dos veces al módulo de memoria durante cada ciclo de instrucción.

Las *líneas de dirección* se utilizan para seleccionar la fuente o el destino de la información que hay sobre el bus de datos. Por ejemplo, si la CPU desea leer una palabra de la memoria, coloca la dirección de dicha palabra en las líneas de dirección. Es evidente que la anchura del bus de direcciones determina la capacidad de direccionamiento de la unidad de memoria del sistema. Además, las líneas de dirección también se pueden utilizar para direccionar los puertos de E/S. Los bits de mayor peso se usan para escoger un módulo particular conectado al bus y los bits de menor peso seleccionan una posición de memoria o un puerto de E/S dentro del módulo. Por ejemplo, en un bus de 8 bits, la dirección 01111111 y anteriores, podrían referenciar a posiciones en un módulo de memoria (módulo 0) con 128 palabras y la dirección 10000000 y posteriores referirse a dispositivos conectados a un módulo de E/S (módulo 1).

Las *líneas de control* gobiernan el uso y el acceso a las líneas de datos y de dirección. Como estas líneas están compartidas por todos los componentes tienen que proveerse de determinados mecanismos que controlen su utilización. Las señales de control transmiten tanto órdenes como información de temporización entre los módulos del sistema. Estas últimas señales indican la validez de la información que hay sobre las líneas de datos y de direcciones. Las órdenes especifican qué operaciones tienen que realizarse. Entre las líneas de control típicas están:

- *Escritura en memoria*: Los datos que están sobre el bus se escriben en la posición direccionada.
- *Lectura de memoria*: El dato contenido en la posición direccionada se coloque sobre el bus.
- *Escritura a E/S*: Los datos que están sobre el bus se sacan al dispositivo de E/S direccionado.
- *Lectura de E/S*: El dato contenido en el dispositivo de E/S direccionado se coloca sobre el bus.
- *Reconocimiento de transferencia*: Los datos han sido aceptados desde el bus o colocados en él.
- *Petición del bus*: Un módulo necesita obtener control del bus.
- *Autorización del bus*: La petición de un módulo para tomar el control del bus se autoriza.
- *Petición de interrupción*: Está pendiente una interrupción.
- *Reconocimiento de interrupción*: La interrupción pendiente ha sido reconocida.
- *Reloj*: Se utiliza para sincronizar operaciones.
- *Reset*: Inicializa todos los módulos.

La operación del bus se puede explicar de forma sencilla de la manera siguiente. Si un módulo desea enviar datos a otro debe hacer dos cosas:

1) Obtener el uso del bus
2) Transferir datos a través del bus

Por otro lado, si lo que un módulo quiere es pedir datos de otro módulo, deberá realizar las siguientes acciones:

1) Obtener el uso del bus
2) Transferir una petición al otro módulo por las líneas de dirección y control apropiadas
3) Esperar a que el otro módulo le envíe los datos

Físicamente, el bus del sistema es una serie de conductores eléctricos en paralelo. Estos conductores son líneas metalizadas depositadas sobre una tarjeta de circuito impreso. El bus se extiende a través de todos los componentes del sistema. En la Figura 1.21 se representa una disposición muy común para la realización de una arquitectura de bus.

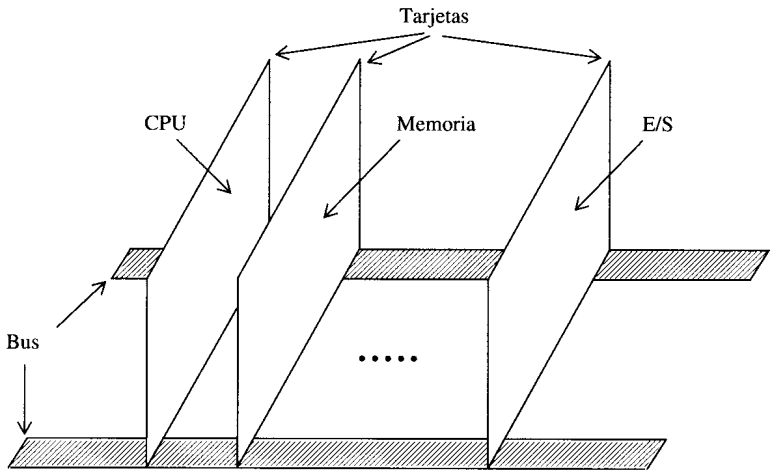


Figura 1.21: Realización física de una arquitectura de bus

En este ejemplo, el bus consiste en dos filas horizontales de conductores. A intervalos regulares a lo largo de las filas, existen puntos de enganche, en forma de ranuras, que se extienden para soportar verticalmente las diferentes tarjetas de circuito impreso.

Cada una de las unidades principales del sistema ocupa una o más tarjetas que se conectan en las diferentes ranuras del bus. La estructura completa se introduce en un chasis. Esta disposición tiene la ventaja de poderse ampliar fácilmente, de manera que se puede adquirir un pequeño sistema y posteriormente expandirlo (más memoria, más unidades de E/S) añadiendo algunas tarjetas. Si falla un componente de una tarjeta, ésta se puede quitar sin dificultad y sustituirla por otra.

1.4.3 Estructura jerárquica de buses

Cuando se aumenta el número de elementos que se conectan al bus su rendimiento puede verse afectado negativamente debido fundamentalmente a tres motivos:

1) El retardo de propagación aumenta cuando se incrementa el número de dispositivos conectados al bus. El tiempo que necesitan las unidades que están compartiendo el bus para coordinarse entre

ellas depende de este retardo de manera que si el control del bus cambia con frecuencia entre los dispositivos puede influir notablemente en su rendimiento.

2) Si el número de solicitudes de transferencias que se van acumulando tiende a la capacidad que tiene el bus se puede producir un atasco similar al que se produce cuando muchos coches a la misma hora se empeñan en utilizar la misma autopista. La forma de solucionar este problema es análoga en ambos casos, en el primero aumentando la velocidad a la que el bus puede enviar sus datos al ampliar su anchura (por ejemplo de 32 a 64 bits), en el segundo incrementando el número de carriles disponibles en la autopista. De todas formas en último término la utilización de un único bus no soluciona el problema de fondo que es que los periféricos que se le conectan requieren una velocidad de transferencia cada vez mayor.

3) Diferente naturaleza de los dispositivos conectados al bus. Así es posible encontrar dispositivos lentos y rápidos, que requieren un uso frecuente o aperiódico, que transmiten grandes o pequeñas cantidades de información, que realizan transferencias por bloques o por caracteres, etc. Todas estas consideraciones hace que sea muy difícil optimizar el bus para todos los casos posibles.

La solución a este problema se encuentra en utilizar dentro de la estructura de interconexión del computador un conjunto de buses organizados jerárquicamente. La Figura 1.22a muestra ejemplos usuales de dispositivos de E/S conectados al bus de expansión. La interfaz *SCSI* (*Small Computer System Interface*) es realmente un bus que se emplea normalmente en los PC's para conectar controladores de disco y otros periféricos. Por su parte la conexión a red permite la posibilidad de interconectarse a redes de área local como por ejemplo Ethernet o a redes de área extendida como Internet.

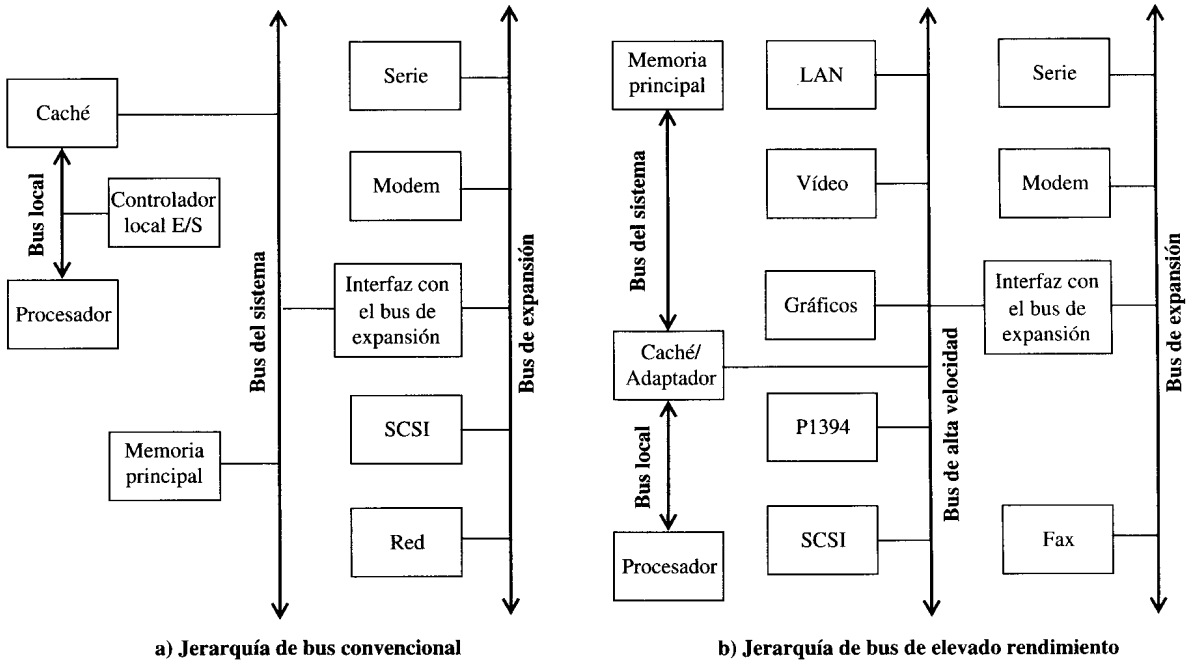


Figura 1.22: Ejemplos de jerarquías de buses múltiples

Este tipo de jerarquía de bus convencional tiene un rendimiento aceptable, aunque comienza a degradarse cuando se conectan dispositivos de E/S que consumen un gran ancho de banda (es decir requieren velocidades de transferencia elevada). La respuesta a este problema ha sido incluir en la estructura de interconexión un bus de alta velocidad que se integra con el resto del sistema mediante un adaptador colocado entre el bus del procesador y el bus de alta velocidad. En la Figura 1.22b se observa un ejemplo de este tipo. Hay también un bus local que conecta el procesador a un controlador de caché que al mismo tiempo está conectado mediante su adaptador correspondiente al bus de alta velocidad que permite conexiones a redes de fibra óptica de 100 Mbits/seg, a controladores especializados en tratamientos gráficos e imágenes y a buses de periféricos como el *SCSI* y el *P1394*. Los dispositivos que poseen menor velocidad se conectan al bus de expansión que emplea una interfaz para compatibilizar el flujo de información entre ambos buses.

Esta estructura con un bus de alta velocidad presenta como ventaja más importante que permite una conexión más estrecha entre el procesador y los periféricos de gran ancho de banda pero manteniéndolos al mismo tiempo independientes de forma que las modificaciones que se produzcan en una de las arquitecturas (procesador o bus de alta velocidad) no afecta a la otra.

1.4.4 Elementos de diseño del bus

Aunque existe una gran diversidad de realizaciones de buses, sólo hay unos cuantos parámetros o elementos de diseño básicos que se utilizan para clasificarlos y diferenciarlos. La Tabla 1.1 relaciona estos elementos.

Tipo	Método de arbitraje	Temporización	Anchura del bus	Transferencia de datos
Dedicado	Centralizado	Síncrona	Dirección	Transferencia de escritura
No dedicado	Distribuido	Asíncrona	Datos	Transferencia de lectura

Tabla 1.1: Elementos de diseño del bus

Tipos de buses

Las líneas del bus se agrupan en dos *tipos* genéricos: *dedicadas* y *no dedicadas*. Una línea de bus dedicada se asigna de forma permanente a una función o a un subconjunto de componentes del computador.

Un ejemplo de dedicación funcional es el uso de líneas de direcciones y de datos independientes, caso frecuente en muchos buses. Sin embargo, no es esencial, ya que la información sobre la dirección y los datos se puede transmitir por el mismo conjunto de líneas, distinguiendo entre ambas posibilidades, mediante una línea de control de *dirección válida.*

Así, al comienzo de una transferencia de datos se coloca la dirección sobre el bus y se activa la línea de dirección válida. En ese momento cada módulo tiene un período de tiempo especificado para copiar la dirección y determinar si es el módulo direccionado. A continuación se quita la dirección del bus y se utilizan las mismas conexiones para la subsiguiente transferencia de lectura o escritura del dato. Este método de utilizar las mismas líneas con objetivos diferentes se conoce como *multiplexación en el tiempo* (también se habla de multiplexación en el tiempo cuando por un bus de N líneas se transmite un dato o una dirección de $2N$ o más bits). La ventaja de la multiplexación temporal radica en que se utilizan menos líneas, esto supone un ahorro de espacio y resulta más económico. Las desventajas son: la necesidad de una circuitería más compleja dentro de cada módulo, y una reducción potencial en el rendimiento del computador puesto que ciertos sucesos que comparten las mismas líneas no pueden tener lugar en paralelo.

La *dedicación física* se refiere al uso de múltiples buses, cada uno de los cuales conecta solamente un subconjunto de módulos. Un caso típico es la utilización de un bus de E/S para interconectar todos los módulos de E/S (Figura 1.14); este bus se conecta con el bus principal mediante algún tipo de módulo adaptador de E/S. La ventaja potencial de la dedicación física es un mayor rendimiento, ya que hay una menor contención del bus. Una desventaja es el aumento del tamaño y del coste del sistema.

Método de arbitraje

En todos los casos, salvo en los sistemas más simples, puede suceder que más de un módulo desee el control del bus. Por ejemplo, un módulo de E/S puede necesitar leer o escribir directamente en memoria sin enviar el dato a la CPU. Como en un instante de tiempo solamente puede transmitir al bus una unidad, se requerirá algún *método de arbitraje* para la utilización de dicho bus. En un sentido amplio los diferentes métodos de arbitraje se pueden clasificar como *centralizados* o *distribuidos*.

En un esquema centralizado, un único dispositivo, denominado el *controlador del bus o árbitro*, es el responsable de asignar el tiempo de utilización del bus. El controlador del bus puede ser un módulo independiente o parte de la CPU. En la Figura 1.23 se muestra de forma elemental la estructura del arbitraje centralizado. En este caso un sólo controlador del bus o árbitro determina que dispositivo tiene acceso. El bus posee una línea de solicitud que puede activarse en cualquier instante de tiempo por uno o varios dispositivos a la vez y las características del sistema vienen dictadas por las siguientes reglas:

1) El árbitro no sabe el número de dispositivos que han solicitado el acceso al bus

2) Sólo es posible distinguir si existen o no solicitudes de acceso al bus

Cuando el árbitro reconoce una *solicitud del bus*, permite su utilización activando la línea de autorización del bus. Esta línea está conectada de forma serie a lo largo de todos los dispositivos. Cuando el módulo que se encuentra más próximo al árbitro recibe la señal de autorización comprueba si la solicitud provenía de él mismo en cuyo caso hace uso del bus e impide que la señal se siga propagando. En el caso contrario, la señal de autorización se transmite al siguiente módulo en la línea que opera de forma análoga y así sucesivamente hasta que algún dispositivo la acepte y acceda a la utilización del bus. Esta estructura de *encadenamiento* (daisy-chaining) tiene la propiedad de que los dispositivos tienen asignadas unas prioridades implícitas en función de su cercanía o lejanía del árbitro. El dispositivo más próximo tiene la prioridad más elevada.

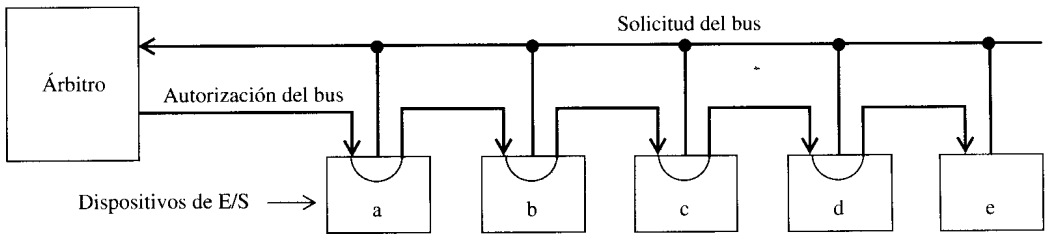


Figura 1.23: Bus con arbitraje centralizado por encadenamiento

Con el fin de mitigar el efecto de las prioridades implícitas que se fundamentan en la proximidad al árbitro, algunos buses disponen de una estructura con múltiples niveles de prioridad donde cada nivel posee sus propias líneas de solicitud y autorización. En la Figura 1.24 se muestra un bus con arbitraje centralizado y

dos niveles de prioridad (lo normal es que los buses tengan 4, 8 o 16 niveles). Cada módulo se encuentra conectado a alguno de los niveles de solicitud del bus. En la Figura 1.24 los módulos *a* y *c* están conectados al nivel de prioridad 1 mientras que los módulos *b*, *d* y *e* emplean el nivel de prioridad 2.

Si de manera simultánea algunos niveles de prioridad solicitan el bus, el árbitro concede su uso al que tiene una mayor prioridad. Finalmente entre los dispositivos que poseen la misma prioridad se aplica la política de encadenamiento que da el acceso al bus a aquel módulo que se encuentra más cerca del árbitro. En la Figura 1.24 en caso de conflicto el módulo *b* supera al *d*, éste al *e* que a su vez supera al módulo *a*. El módulo *c* tiene la menor prioridad porque se encuentra al final de la cadena con prioridad más baja.

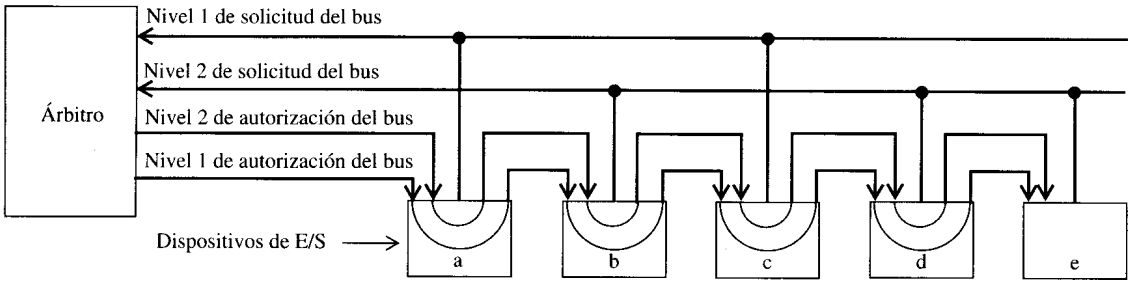


Figura 1.24: Bus con arbitraje centralizado y dos niveles de prioridad

En un esquema distribuido, no hay un controlador central. En su lugar cada módulo contiene la lógica de control suficiente para poder acceder al bus y todos ellos actúan de forma cooperativa para compartir el recurso. En esencia el objetivo de ambos métodos de arbitraje (centralizado y distribuido) es designar a un dispositivo, que bien puede ser la CPU o un módulo de E/S, como *maestro*.

El maestro o árbitro puede iniciar una transferencia de datos (por ejemplo leer o escribir) con algún otro dispositivo que actúa como *esclavo* para este intercambio particular. En la Figura 1.25 se muestra un esquema de un bus con arbitraje distribuido que sólo emplea tres líneas independientemente del número de módulos que estén conectados. La primera línea es la de *solicitud del bus*. La segunda es la de *ocupada* que se activa por el módulo maestro correspondiente. La tercera es la línea de *arbitraje del bus* que recorre todos los módulos mediante encadenamiento. Un extremo de esta cadena se conecta a una fuente de alimentación con el fin de mantener la línea de arbitraje siempre activa.

Cuando ningún módulo solicita el bus, la línea activa de arbitraje se propaga a través de todos los módulos. Para que un módulo cualquiera pueda utilizar el bus lo primero que hace es comprobar si se encuentra libre y si la señal de arbitraje *Ent* que recibe a su entrada está activa, si no lo está no puede convertirse en maestro del bus. Sin embargo si *Ent* se encuentra activa, el módulo desactiva su señal de salida *Sal* que es la que necesitan el resto de los módulos que se encuentran en la cadena para realizar la misma operación.

Como resultado de este proceso, sólo un módulo de la cadena tendrá activada su señal *Ent* y desactivada su señal *Sal* con lo que pasa a ser el maestro del bus. En ese momento activa las líneas ocupada y su salida *Sal* e inicia su transferencia. Con esta estrategia se puede observar que el módulo situado más a la izquierda en la cadena que precise el bus es el que lo consigue, de manera que este esquema es análogo al de un bus con arbitraje centralizado por encadenamiento (ver Figura 1.23) pero con la ventaja de que no requiere un árbitro por lo que es más barato, rápido y fiable al no estar sujetos a posibles mal funcionamiento del mismo.

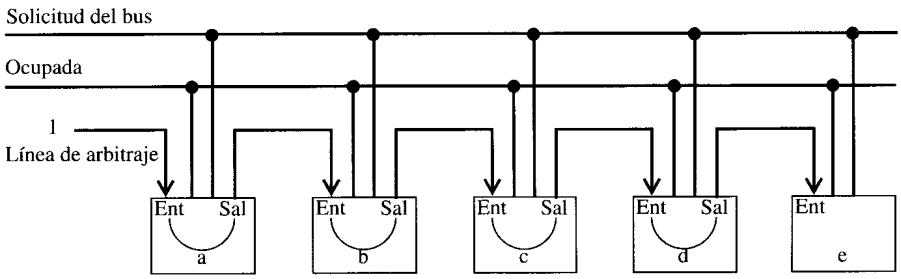


Figura 1.25: Bus con arbitraje distribuido

Temporización

La *temporización* contempla la forma en cómo se coordinan los sucesos sobre el bus. En el caso de la *temporización síncrona* la ocurrencia de sucesos sobre el bus está determinada por un reloj. El bus incluye una línea de reloj sobre la que se transmite una secuencia regular de unos y ceros alternativos de igual duración. Una simple transmisión 1-0 se conoce como *ciclo de reloj o ciclo del bus* y define una ranura de tiempo. Los otros dispositivos conectados al bus pueden leer la línea de reloj, y comienzan la operación al principio de un ciclo del bus. La Figura 1.26 muestra el diagrama de temporización para una operación de lectura síncrona. Otras señales del bus pueden cambiar con el flanco de subida de la señal de reloj (con un ligero retardo en la respuesta). La mayoría de los sucesos ocupan un único ciclo de reloj. En este ejemplo sencillo la CPU emite una señal de lectura y coloca una dirección de memoria en el bus de direcciones. También emite una señal de comienzo para marcar la presencia de la dirección y de información de control sobre el bus. Un módulo de memoria reconoce la dirección y después de un retardo de un ciclo coloca el dato y una señal de reconocimiento sobre el bus.

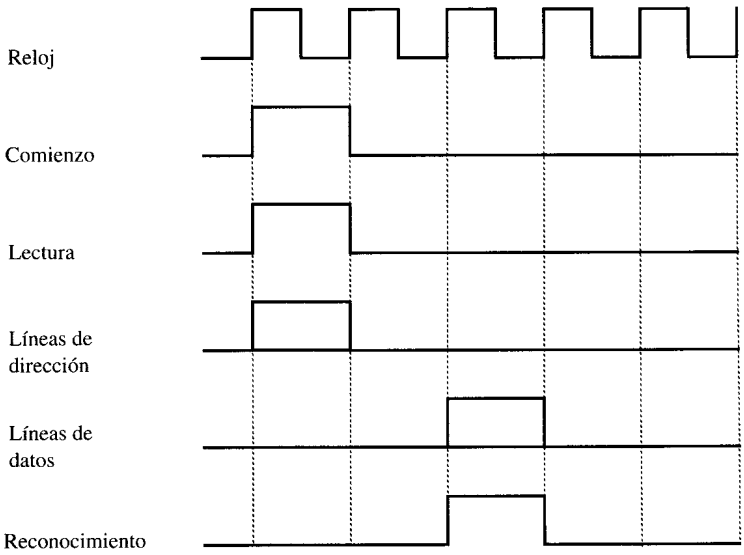


Figura 1.26: Temporización síncrona

Con la *temporización asíncrona* la ocurrencia de un suceso sobre el bus sigue a, y depende de, la aparición de un suceso previo sin que tenga que producirse en un instante concreto simultáneamente con un flanco de la señal de reloj. En el ejemplo de la Figura 1.27 (lectura de una posición de memoria por el procesador) la CPU coloca la dirección y una señal de lectura sobre el bus. Después de una pausa para que estas señales se estabilicen, emite una señal *MSIN* (maestro sincronizado) que indica la presencia de una dirección y señales de control válidas. Cuando el dispositivo esclavo, que en este caso es el módulo de memoria, ve esta señal realiza su trabajo tan rápido como puede, activando al terminar la señal *ESIN* (esclavo sincronizado).

Tan pronto como el maestro ve activada la señal *ESIN* sabe que el dato está disponible de modo que lo almacena en un registro interno de la CPU y desactiva la línea *MSIN*. A su vez, cuando el esclavo observa que está desactivada la señal *MSIN*, sabe que se ha finalizado la transferencia así que desactiva *ESIN*, las líneas de datos, de lectura y de dirección, y retorna a la situación original, con todas las señales desactivadas y aguardando al siguiente ciclo. En los cronogramas de los buses asíncronos se emplean flechas para indicar las relaciones de causa-efecto tal como se muestra en la Figura 1.27. Al conjunto de señales enlazadas de esta forma se le denomina *sincronización completa o bidireccional*, cuya parte básica consta de las cuatro acciones siguientes:

1) Se activa *MSIN*
2) Se activa *ESIN* en respuesta a *MSIN*
3) Se desactiva *MSIN* en respuesta a *ESIN*
4) Se desactiva *ESIN* en respuesta a la desactivación de *MSIN*

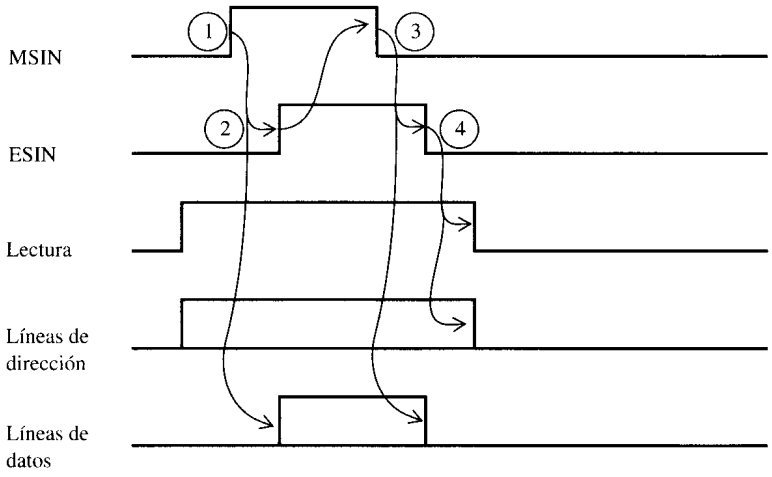


Figura 1.27: Temporización asíncrona

La temporización síncrona es más simple de realizar y verificar, sin embargo, es menos flexible que la temporización asíncrona. Este último hecho se debe a que todos los dispositivos en un bus síncrono están funcionando con una frecuencia de reloj fija, de manera que el sistema no puede beneficiarse de los avances que se produzcan en el rendimiento de sus diferentes dispositivos. Aún más, si un bus con temporización síncrona tiene conectado un grupo heterogéneo de dispositivos, de diferentes velocidades, el bus debe ade-

cuarse al más lento impidiendo que los más rápidos empleen todo su potencial. Con la temporización asíncrona pueden aprovechar mejor el bus una mezcla de dispositivos lentos y rápidos de diferentes tecnologías.

Anchura del bus

El concepto de *anchura del bus* se presentó ya en el apartado 1.4.2 al comentar la estructura del bus. El ancho del bus de datos tiene influencia sobre el rendimiento del sistema, así cuanto mayor sea más cantidad de información se puede transmitir simultáneamente. Por su parte el ancho del bus de direcciones modifica la capacidad del sistema ya que cuanto más ancho sea, mayor es el número de posiciones de memoria que se pueden referenciar de forma directa.

Transferencia de datos

Todos los buses permiten los dos tipos de transferencia siguientes:

1) *Transferencia de escritura.* En esta transferencia el dato lo envía el maestro y lo recibe el esclavo

2) *Transferencia de lectura.* En esta transferencia el dato lo envía el esclavo y lo recibe el maestro

La transferencia es diferente según posea o no el bus multiplexación temporal de direcciones y datos. En el primer caso el bus se utiliza primero para especificar la dirección y luego para enviar el dato. En el segundo caso, cuando se dispone de líneas específicas para datos y direcciones, se coloca la dirección en el bus de direcciones y se mantiene mientras el dato se sitúa en el bus de datos.

Se pueden considerar también algunos tipos de operaciones combinadas. Así por ejemplo una operación de *lectura-modificación-escritura* consiste en una lectura a la que sigue de forma inmediata una escritura en la misma dirección. La dirección se da una única vez al comienzo de la operación. Toda la operación se considera indivisible por lo que no es posible que otros posibles maestros del bus puedan acceder al dato. Esta característica tiene interés cuando se desea proteger los recursos de memoria compartida en un sistema que admite multiprogramación.

Otra operación de carácter también unitario es la de *lectura después de escritura* que consiste en una escritura seguida inmediatamente de una lectura en la misma dirección. Esta característica es de utilidad cuando se desea verificar el resultado de una operación de escritura.

Determinados buses permiten *transferencias de bloques de datos*. En este tipo de buses a un ciclo de dirección le siguen un determinado número de ciclos de datos. El primer dato se envía hacia o desde la dirección especificada; el resto de los datos se transfieren a partir de las direcciones siguientes. Al estudiar en la sección 3-5 el mecanismo de acceso directo a memoria se considerarán transferencias de esta clase.

1.4.5 Consideraciones prácticas en la conexión mediante bus

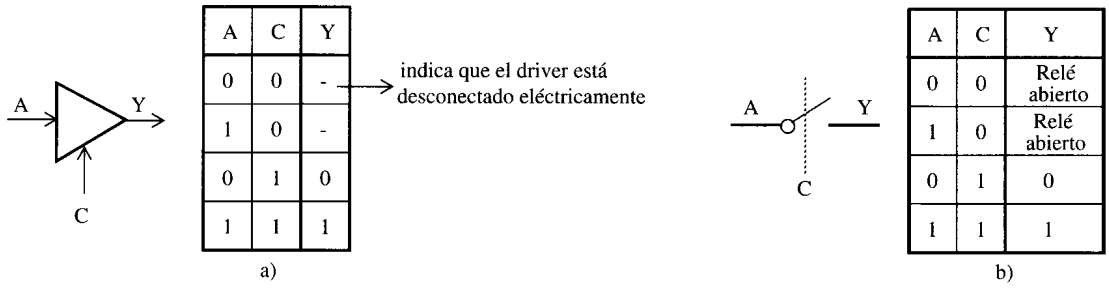
La utilización de un bus compartido por las distintas unidades del computador (CPU, memoria y E/S) implica el empleo de unos circuitos de acoplo para garantizar el funcionamiento correcto del bus. Cada línea del bus debe conectarse a un número de entradas variable y normalmente elevado, lo que puede provocar problemas de *fan out*. El fan out de un circuito lógico especifica el número máximo de entradas a las que puede conectarse su salida sin que se produzca ninguna anomalía. Este número depende de la tecnología de fabricación de los circuitos integrados que se utilicen. Así por ejemplo, para evitar los problemas de fan out, antes de conectar una línea del bus de dirección a todos los circuitos integrados de un bloque de memoria se la hace pasar a través de una etapa de circuitos denominados "drivers", los cuales no realizan ninguna función lógica y tienen como misión conseguir que desde el bus de dirección todo el bloque de memoria se contemple

como un único circuito a efectos de fan out. Además, permiten que las diferentes unidades del computador compartan la utilización del bus. Estos drivers se realizan con un tipo particular de puertas lógicas denominadas *puertas triestado.*

Puertas Triestado

Son circuitos lógicos que disponen de una señal de control que permite la aparición en su salida de tres estados diferentes. En la Figura 1.28a se representa el diagrama de bloques de un driver triestado y su tabla de operación. Tiene dos entradas A (entrada del dato) y C (entrada de control) y una salida Y. Cuando la entrada de control C está activa ($C = 1$), permite al driver funcionar conectando la entrada con la salida ($Y = A$) que presenta los dos estados lógicos característicos: 0 y 1. Si la señal de control C se desactiva ($C = 0$), la puerta se desconecta de su salida Y. En este caso se dice que la salida está en un *tercer estado* (de ahí el nombre de puerta triestado) *de alta impedancia.*

Se puede ver la puerta triestado como un relé (ver Figura 1.28b) de manera que cuando $C = 0$, el relé está abierto y su entrada A está desconectada de su salida Y, y cuando $C = 1$ el relé está cerrado y la salida Y se conecta a la entrada A ($Y = A$).

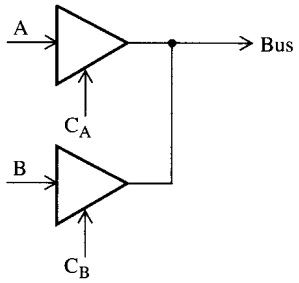


A	C	Y
0	0	-
1	0	-
0	1	0
1	1	1

A	C	Y
0	0	Relé abierto
1	0	Relé abierto
0	1	0
1	1	1

Figura 1.28: Funcionamiento de un driver triestado

El punto importante a tener en cuenta es que, si algunos drivers triestado tienen conectadas sus salidas a una misma línea de un bus (ver Figura 1.29) sólo puede estar activa la señal de control de uno de los drivers.



C_A	C_B	Bus
0	0	Alta impedancia
1	0	A
0	1	B
1	1	Función erronea

Figura 1.29: Conexión de 2 puertas triestado a una línea de bus común

En la Figura 1.30 se muestra un driver triestado que invierte su entrada. Cuando $C = 0$ el driver está desconectado de su salida y cuando $C = 1$ entonces $Y = \overline{A}$ (la salida es el complemento de la entrada).

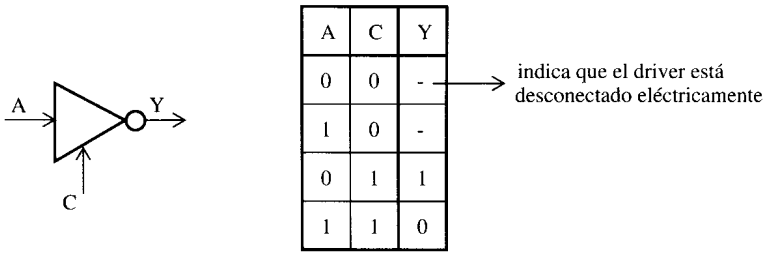


Figura 1.30: Driver triestado con salida invertida

En la Figura 1.31 se representa el esquema de un circuito triestado de conexión unidireccional con señal de control común (cada una de las líneas de entrada X_i, $i = 1, 2, ...m$ lleva asociada un driver triestado).

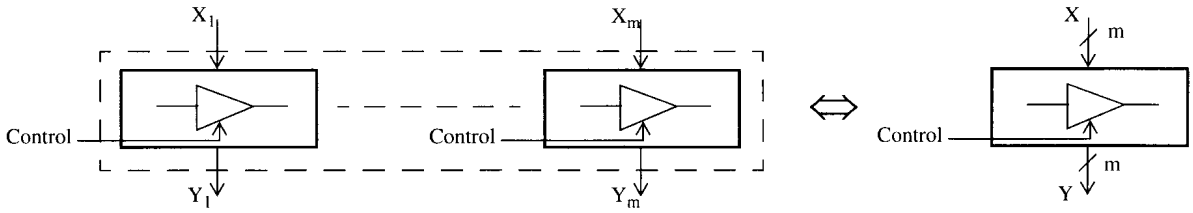


Figura 1.31: Circuito triestado de conexión unidireccional con control común

Como ejemplo de uso de este circuito, se puede establecer la comunicación mediante bus compartido entre 3 registros fuente (R_1, R_2, R_3) y 2 registros destino (R_4, R_3) todos de 4 bits (ver Figura 1.32).

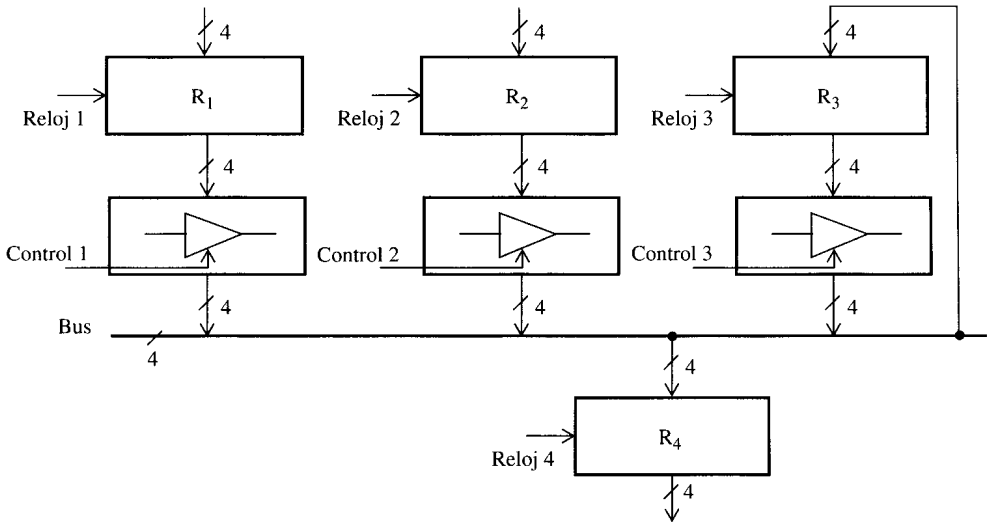


Figura 1.32: Ejemplo de circuito de comunicación mediante bus compartido

Uno de los registros (R_3) actúa como fuente y destino. Las transferencias de información entre los registros representados en la Figura 1.32 se realizan según sea el valor de las señales *Control 1*, *Control 2*, *Control 3* y las correspondientes señales de reloj en los registros. De acuerdo con lo dicho sobre el funcionamiento de los drivers triestado, en este ejemplo no puede haber simultáneamente dos señales de control que tengan el valor lógico 1. En la Figura 1.33 se muestra el cronograma de la transferencia del contenido del registro R_1 al registro R_3.

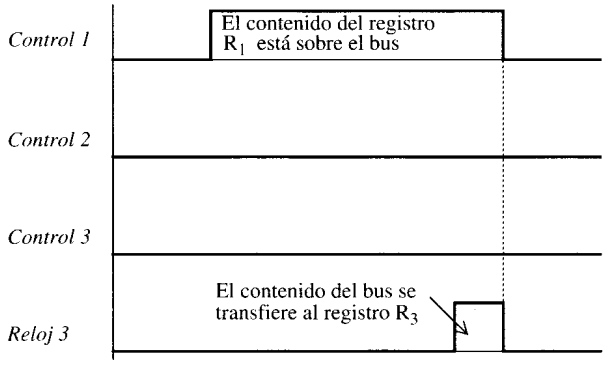
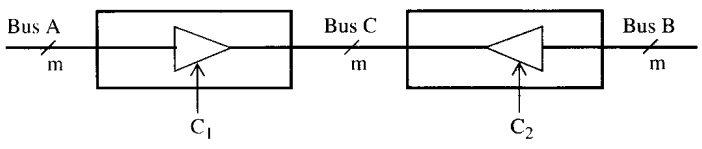


Figura 1.33: Cronograma de la transferencia del registro R_1 al registro R_3

Los circuitos triestado también se utilizan cuando las necesidades impuestas por problemas de carga en un bus (número de unidades conectadas, excesiva longitud, etc) obligan a expandirlo. Esto fuerza a incluir una circuitería de recuperación y de reenvío de la señal (repetidores). En este caso hay que hacer una distinción según se trate de buses unidireccionales o bidireccionales.

a) *Bus unidireccional*

Tal como se muestra en la Figura 1.34, la inclusión de circuitos triestado permite seleccionar los instantes de paso de información e incluso aislar una parte del bus.

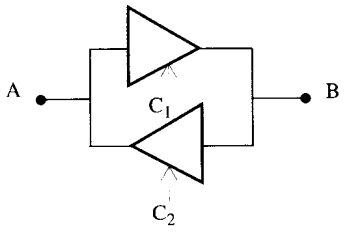


C_1	C_2	Bus C
0	0	Desconectado
1	0	Bus A
0	1	Bus B
1	1	Error 2 cargas al bus

Figura 1.34: Conexión unidireccional

b) *Bus bidireccional*

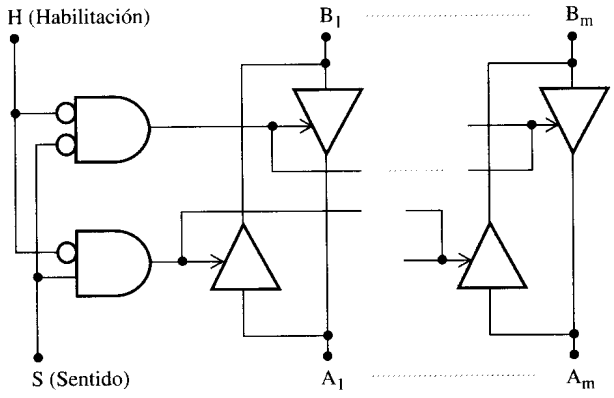
Si se utiliza un bus bidireccional su ampliación lleva consigo el empleo, en forma selectiva, de circuitos triestado para las dos direcciones (ver Figura 1.35).



C_1	C_2	Función
0	0	Aislados A y B
1	0	A → B
0	1	B → A
1	1	Error, [illegible] oscilaciones

Figura 1.35: Circuito triestado bidireccional

Un ejemplo de este tipo de circuitos de conexión bidireccional de *m* bits se representa en la Figura 1.36.



H	S	Operación
0	0	B → A
0	1	A → B
1	0	Aislado
1	1	Aislado

Figura 1.36: Circuito de conexión bidireccional con señal de control común

En la Figura 1.37 se representa de forma compacta el circuito de la Figura 1.36 con señales de control comunes: *S* (sentido de la comunicación) y *H* (habilitación del circuito).

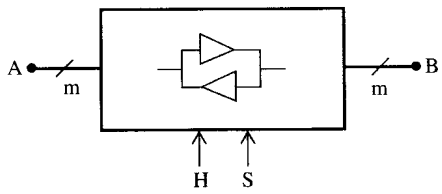


Figura 1.37: Representación simbólica del circuito de conexión bidireccional

Cuando se amplía un bus bidireccional, para las transferencias de información entre los dispositivos conectados es muy importante que las señales de control se activen en la secuencia correcta. Si esto no se tiene en cuenta puede suceder que la expansión perturbe al resto del circuito. En la Figura 1.38 se ilustra con un ejemplo este problema.

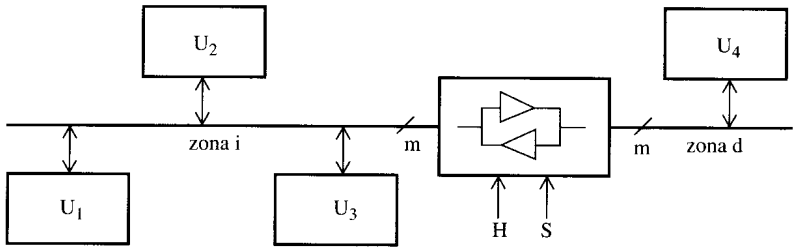


Figura 1.38: Ampliación de un bus bidireccional

En la Tabla 1.2 se resume como hay que aplicar la secuencia de señales de control en diferentes casos. Se emplea la siguiente notación: $U_i \leftarrow U_j$ significa transferir el contenido de la unidad j a la unidad i (en otros términos quiere decir que desde la unidad i se lee la unidad j).

$U_j \leftarrow U_4$, j = 1, 2, 3	$U_j \leftarrow U_i$, i, j =1, 2, 3 (i ≠ j)	$U_4 \leftarrow U_j$, j = 1, 2, 3
- Salida de U_4 al bus - H = 0 (modo activado) - S = 0 (conexión d → i)	- Salida de U_2 o U_3 al bus - H = 1 (modo desactivado)	- Salida de U_j al bus - H = 0 (Modo activado) - S = 1 (conexión i → d)

Tabla 1.2: Secuencia de las señales de control

En la segunda columna es necesario poner la señal de habilitación H en modo desactivado (H = 1), porque si no estaría saliendo a la zona d del bus el contenido de U_2 o U_3 desde la zona i del bus que en ese momento puede estar realizando otra transferencia.

1.4.6 Ejemplo de estructura de bus: El Unibus

El *Unibus* lo introdujo DEC (*Digital Equipment Corporation*) en 1970 para su familia de computadores PDP 11. Es un bus simple al que se conectan todos los componentes del sistema. En la Tabla 1.3 se da una relación de las 56 líneas que lo constituyen. El Unibus utiliza un esquema de arbitraje asíncrono y centralizado. Unido a la CPU hay un árbitro cuya tarea es autorizar las peticiones de acceso al control del bus. La Figura 1.39 muestra, de forma simplificada, el mecanismo de funcionamiento.

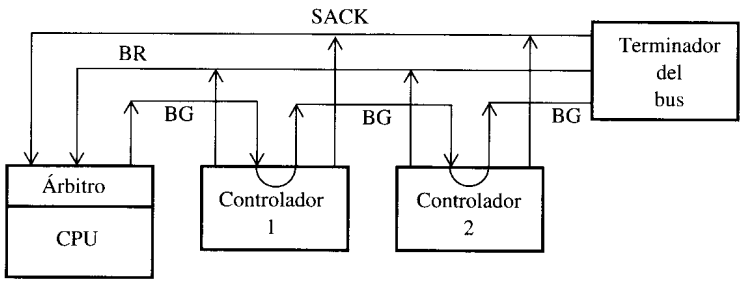


Figura 1.39: Diagrama simplificado del mecanismo de arbitraje

Los módulos de E/S llamados *controladores* se conectan a una línea común de petición del bus (*BR*). Cuando un controlador desea tener el control del bus solicita permiso y para hacerlo pone *BR* (línea de petición del bus) a 1. El árbitro da el permiso colocando a su vez la señal *BG* (línea de autorización del bus) a 1. Como varios controladores comparten la misma línea *BR*, se necesita algún mecanismo para determinar a cual de ellos se le da el control. Esto lo hace la línea *BG* mediante un esquema de encadenamiento. Tal como se puede ver en la Figura 1.39 la línea *BG* termina y comienza en cada controlador. Si un controlador tiene *BR* a *0*, responderá a una entrada de *BG* = 1 colocando también su salida *BG* a 1.

Designación	Líneas	Descripción
Transferencia de datos		
Dirección	18	Dirección de memoria o dispositivo de E/S
Datos	16	Dato a transferir
Control (C_1, C_0)	2	Tipo (byte, palabra) y sentido de transferencia
Paridad (PA, PB)	2	Indica error de paridad
Sincronización del maestro	1	El maestro indica al esclavo que hay información válida en el bus
Sincronización del esclavo	1	El esclavo indica que ha leído o que ha puesto un dato en el bus
Control del bus		
Interrupción	1	Confirmación del módulo que va a interrumpir
Petición del bus (BR7-BR4)	4	Petición del bus para una interrupción
Autorización del bus (BG7-BG4)	4	La CPU indica que ha sido concedida una petición del bus
Petición de no procesador (NPR)	1	Un dispositivo DMA solicita control para una transferencia de datos
Autorización de no procesador (NPG)	1	Confirma que se autoriza NPR
Reconocimiento de selección (SACK)	1	El dispositivo que solicita el bus reconoce la recepción de la autorización
Bus ocupado (BBSY)	1	El bus está siendo utilizado por el Maestro
Inicialización		
Inicializar	1	Inicializa todos los dispositivos periféricos
Línea AC baja	1	Indica que va a ocurrir un fallo de potencia
Línea DC baja	1	Suministro de potencia está fuera de límites

Tabla 1.3: Líneas del bus Unibus

Si por el contrario el controlador tiene *BR* = 1, entonces no pondrá su salida *BG* a 1, pero aceptará la entrada *BG* como señal de que se le ha autorizado el control del bus. El efecto de este esquema es proporcionar una prioridad más alta a aquel controlador que está físicamente más próximo a la CPU. En la Figura 1.40 se muestra la secuencia de temporización que tiene lugar. Se supone que el *controlador 1* o la

CPU está actuando como maestro del bus y que el *controlador 2* desea obtener el control:

1) El *controlador 2* pone *BR* a 1. Esto indica al árbitro que está pendiente una petición del bus.
2) Pasado un cierto tiempo (indeterminado para el controlador), el árbitro autoriza el control colocando *BG* a 1. Sin embargo, el *controlador 2* no puede todavía tomar el control puesto que el bus está ocupado (*BBSY* a 1) y por lo tanto debe esperar hasta que finalice la operación que está en curso.
3) El *controlador 2* reconoce que ha sido informado de estar seleccionado emitiendo una señal de reconocimiento de selección (*SACK* a 1).
4) El *controlador 2* ahora puede quitar su orden de petición del bus (*BR* a 0).
5) Cuando el árbitro recibe la señal *SACK*, pone *BG* a 0. *SACK* permanece en el nivel 1, indicando así que una petición que se ha autorizado todavía está pendiente. Mientras persista esta situación el árbitro no aceptará nuevas peticiones de acceso al bus y estará por consiguiente desactivado.
6) Después de un cierto tiempo (no prefijado de antemano), el *controlador 1* finaliza su operación y pone la señal *BBSY* a 0, lo que indica que ha dejado libre el bus.
7) En este momento el *controlador 2* puede conseguir el control del bus colocando *BBSY* a 1.
8) Una vez que ha obtenido el control del bus, el *controlador 2* pone *SACK* a 0, lo que activa de nuevo al árbitro para poder recibir una nueva petición.

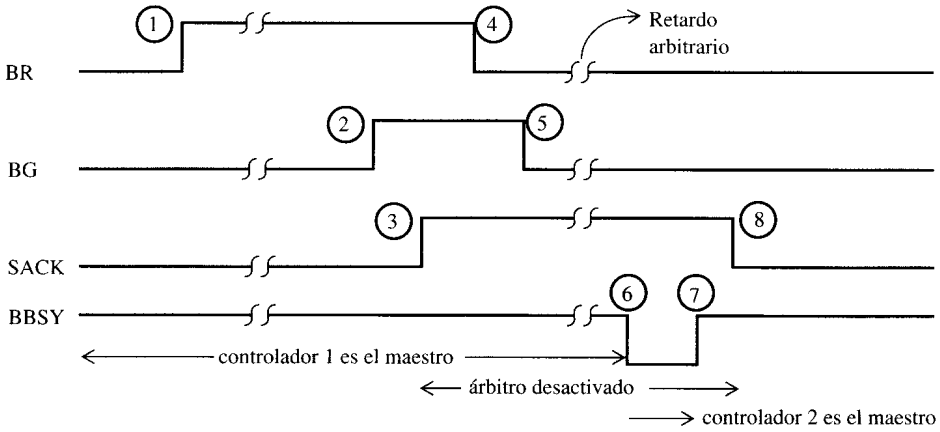


Figura 1.40: Temporización del mecanismo de arbitraje

El Unibus permite cinco grupos de controladores. Cada grupo se conecta a su propia línea de petición del bus (*BR*4, *BR*5, *BR*6, *BR*7, *NPR*) que está encadenada a su línea de autorización del bus (*BG*4, *BG*5, *BG*6, *BG*7, *NPG*) tal como se muestra en la Figura 1.39. Los cinco grupos comparten una misma línea *SACK*. El objetivo de estos grupos es asignar distintos niveles de prioridad a diferentes conjuntos de dispositivos periféricos. De esta forma, si dos controladores en dos niveles de prioridad distintos solicitan el control del bus, el árbitro se lo concederá a aquel peticionario que tenga mayor prioridad. Aún más, en cualquier instante de tiempo si la CPU está operando con un nivel de prioridad dado, solamente se autorizarán peticiones para conseguir el control del bus que tengan una prioridad más elevada.

La prioridad de ejecución (*PRI*) de la CPU del PDP-11 varía desde 0 (la más baja) hasta 7 (la más alta). Normalmente, el programa del usuario opera con prioridad 0. El sistema operativo puede funcionar con prioridades más altas. Por ejemplo, si la CPU está procesando una interrupción de un dispositivo con prioridad 6 entonces está operando con ese nivel de prioridad. El árbitro autoriza el control del bus a módulos diferentes de la CPU de acuerdo con las siguientes reglas:

1) Siempre se autoriza una petición de no-procesador (*NPR*), mediante confirmación a través de *NPG*. La petición es para permitir que un controlador de E/S efectúe una transferencia de acceso directo a memoria.

2) Siempre que la CPU está en la fase de interrupción de un ciclo de instrucción (Figura 1.8) realiza lo siguiente:

   a) Si *PRI* < 7 y *BR7* = 1 entonces *BG7* = 1, si no

   b) Si *PRI* < 6 y *BR6* = 1 entonces *BG6* = 1, si no

   c) Si *PRI* < 5 y *BR5* = 1 entonces *BG5* = 1, si no

   d) Si *PRI* < 4 y *BR4* = 1 entonces *BG4* = 1

Todas las transferencias de datos en el Unibus se realizan con la misma temporización. En cada caso hay un maestro y un esclavo. El maestro efectúa o una operación de salida de datos (escribir al esclavo) o una de entrada (leer del esclavo). Posibles combinaciones de maestro-esclavo son:

- *Maestro = CPU; Esclavo = Memoria.* Ejecución de una instrucción de lectura o escritura en memoria.
- *Maestro = CPU; Esclavo = E/S.* Ejecución de una instrucción de E/S.
- *Maestro = E/S; Esclavo = Memoria.* Un módulo de E/S está accediendo directamente a memoria para efectuar una transferencia de datos entre una unidad externa y memoria.

En la Figura 1.41 se da la temporización de la secuencia de sucesos para la operación de salida de datos:

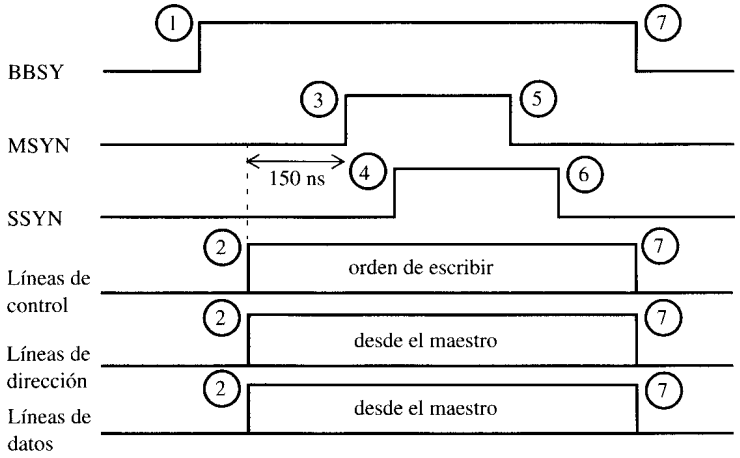


Figura 1.41: Temporización para salida de datos

1) Una vez que se le ha autorizado al maestro el control del bus (ver Figura 1.41) comienza la operación colocando *BBSY* a 1.

2) El maestro coloca la dirección de memoria deseada o de E/S sobre las líneas de dirección y el dato que se va a escribir en las líneas de datos. También asigna una combinación de líneas de control para indicar que está en marcha una orden de escritura.

3) El maestro realiza una pausa de 150 nseg, de los que 75 nseg corresponden al caso más desfavorable de retardo de propagación de la señal a lo largo del bus, y los otros 75 nseg para dar tiempo a que todos los esclavos comprueben y comparen las líneas de dirección. A continuación pone la línea *MSYN* (sincronización del maestro) a 1 para indicar que hay información válida en el bus.

4) Las líneas de dirección deberían coincidir con una de las direcciones de un módulo de memoria o de algún módulo de E/S. El módulo apropiado responde leyendo y almacenando las señales que hay en las líneas de datos. Cuando ha capturado el dato coloca *SSYN* (sincronización del esclavo) a 1.

5) El maestro cuando reconoce la señal *SSYN*, pone *MSYN* a 0.

6) El esclavo pone *SSYN* a 0.

7) El maestro finaliza la operación colocando *BBSY*, las líneas de datos, de dirección y de control a *0*.

De forma análoga se realiza la operación de entrada de datos. En la Figura 1.42 se muestra el diagrama de temporización correspondiente.

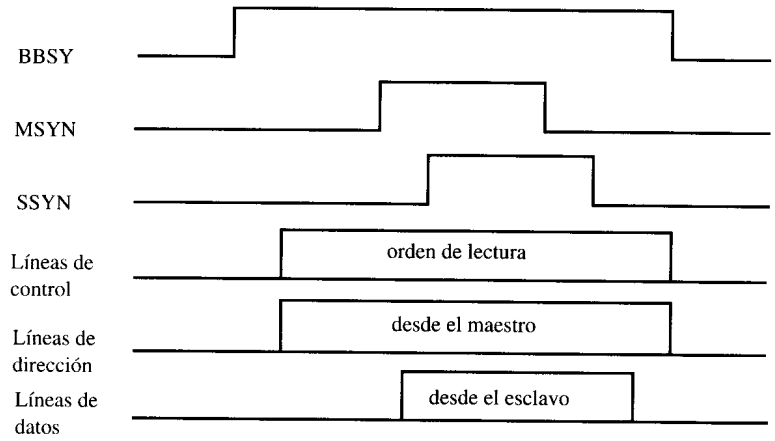


Figura 1.42: Temporización para entrada de datos

Sección 1-5

Conclusiones

Un computador es un sistema complejo que puede llegar a estar constituido por millones de componentes electrónicos elementales. ¿Cómo podemos describirlos de una forma clara y concisa?. La clave está en reconocer la estructura jerárquica de la gran mayoría de los sistemas complejos incluyendo entre ellos al computador.

Esta naturaleza multinivel de los sistemas complejos es esencial para comprender tanto su descripción como su diseño. En cada nivel se ha de analizar su estructura y su función en el sentido siguiente:

- *Estructura*: La forma en que se interrelacionan las componentes
- *Función*: La operación de cada componente individual como parte de la estructura

A partir de estas consideraciones generales, en este tema se ha analizado el funcionamiento de un computador considerado como una entidad de nivel superior que consta de un procesador, memoria y unidades de E/S.

En primer lugar se ha visto, de forma resumida, los diversos componentes de un computador y sus requisitos de entrada-salida. Desde un punto de vista operativo, la conducta funcional de un computador consiste en el intercambio de datos y señales de control entre sus diversos componentes. Para poder soportar estos intercambios las distintas unidades de un computador deben interconectarse de alguna forma.

A continuación se consideraron los principales enfoques estructurales para la interconexión, incluyendo consideraciones temporales y los protocolos de arbitraje en el intercambio de información entre componentes.

Por su particular interés e importancia se dedicó una atención especial a la estructura de interconexión tipo bus. Las líneas de un bus se pueden clasificar en tres grupos funcionales: líneas de datos, líneas de dirección y líneas de control. Desde el punto de vista del diseño de un bus existen una serie de parámetros que permiten sistematizar su estudio y clasificarlos. Estos parámetros son los siguientes:

- *Tipos de buses*. De líneas dedicadas o multiplexadas en el tiempo
- *Método de arbitraje*. Centralizado o distribuido
- *Temporización*. Síncrona o asíncrona
- *Anchura del bus*. De datos, dirección y control
- *Transferencia de datos*. De lectura o escritura

Sección 1-6

Problemas

1) La máquina hipotética de la Figura 1.5 tiene también dos instrucciones de E/S:

a) 0011 = Cargar acumulador (AC) desde E/S

b) 0111 = Almacenar acumulador a E/S

En estos casos, la dirección de 12 bits identifica un dispositivo concreto de E/S. Mostrar la ejecución (utilizando el formato de la Figura 1.6) para el siguiente programa:

1) Cargar AC desde el dispositivo 7

2) Sumar el contenido de la posición de memoria 850_{16}

3) Almacenar el contenido del AC en el dispositivo 5

2) Un computador contiene un módulo de E/S que controla un teletipo simple formado por un teclado y una impresora. En la CPU y conectados directamente con el bus del sistema están contenidos los siguientes registros:

a) *INPR*: Registro de entrada (8 bits)

b) *OUTR*: Registro de salida (8 bits)

c) *FGI*: Flag de entrada (1 bit)

d) *FGO*: Flag de salida (1 bit)

e) *IEN*: Interrupción activa (1 bit)

Tanto las entradas desde el teclado como la salida de la impresora del teletipo están controladas por el módulo de E/S. El teletipo es capaz de codificar un símbolo alfanumérico en una palabra de 8 bits y a la inversa de decodificar una palabra de 8 bits en un símbolo alfanumérico. Se pide:

a) Describir cómo la CPU, utilizando los cuatro registros (*INPR, OUTR, FGI y FGO*) descritos, puede realizar una operación de E/S con el teletipo.

b) Describir cómo se puede efectuar la operación, de forma más eficaz, usando también el registro *IEN*.

3) La memoria de un computador consta de una serie de módulos conectados a un bus de memoria común. Cuando se realiza una petición de escritura, el bus está ocupado por las señales de datos, dirección y control durante 100 nseg. En esos mismos 100 nseg y en los 500 nseg siguientes, el módulo de memoria

direccionado ejecuta 1 ciclo de lectura aceptando y almacenando el dato. Las operaciones de los módulos de memoria pueden solaparse pero sólo puede haber una petición en un instante de tiempo.

a) Si se supone que hay 8 módulos de memoria conectados al bus, ¿cuál es la máxima velocidad posible (en palabras/segundo) con la que se pueden almacenar los datos?

b) Representar gráficamente la velocidad máxima de escritura en función del tiempo de ciclo del módulo, suponiendo que se tienen 8 módulos y un tiempo de ocupación del bus de 100 nseg.

4) Para conseguir un ahorro en el número de puertas lógicas utilizadas, a menudo, los buses se multiplexan en el tiempo. Se considera un computador con una palabra de 48 bits, un tiempo de ciclo de la memoria principal de 600 nseg y una velocidad de transferencia desde el disco de 10^7 bits/seg. Se supone que cada transmisión sobre el bus necesita 750 nseg para enviar el dato y las diferentes órdenes del protocolo de comunicación.

a) ¿Cuántos bits de datos se enviarían en cada período de 750 nseg y qué formato del bus se escogería?

b) ¿Qué fracción del ancho de banda de la memoria principal se gasta en una operación de E/S del disco?

c) Representar el cronograma de los sucesos que tienen lugar durante la transmisión de una operación de entrada continua desde el disco a la memoria principal.

5) El bus de E/S de un computador tiene un mecanismo de arbitraje por encadenamiento (daisy chain) y tiene conectados k dispositivos (d_1, d_2, ..., d_k). Cada dispositivo d_i utiliza una fracción α_i de la capacidad del bus:

$$0 < \alpha_i < 1; \qquad \text{siendo} \qquad \sum_{i=1}^{k} \alpha_i \; < 1$$

que tiene un valor normalizado de 1.

a) ¿Cómo se deberían conectar los dispositivos para conseguir el máximo de la capacidad media sobrante?. Se define la capacidad sobrante de un dispositivo como 1 menos la suma de las capacidades utilizadas por todos los dispositivos que tienen una prioridad superior.

b) Comentar lo que significan esos resultados y dar un ejemplo de cuándo no se pueden aplicar.

6) Considérense la disposición de las líneas de petición del bus (*BR*) y de autorización del bus (*BG*) mostradas en la Figura 1.43.

a) ¿Cómo debería el dispositivo *i* generar *BR*?

b) Interpretar la entrada *BGN*

c) Generar la salida *BGO*

d) Especificar el cronograma de las señales y la prioridad de cada dispositivo

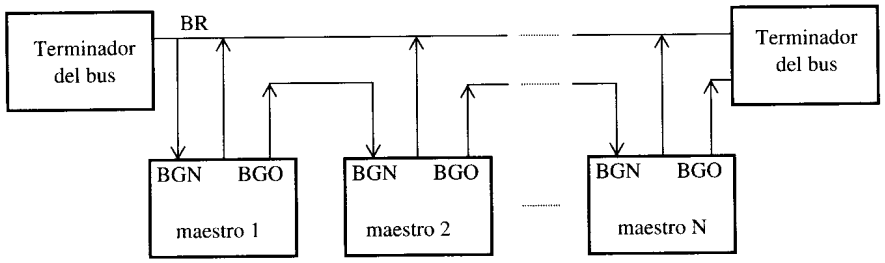


Figura 1.43: Esquema de arbitraje del bus descentralizado

Tema 2

Unidad de memoria

El objetivo de este tema es el estudio del elemento de almacenamiento de información de un computador: la *unidad de memoria*. Como la gran mayoría de los computadores son del tipo de programa almacenado, la unidad de memoria contiene las instrucciones que ejecuta y los datos sobre los que han de actuar dichos programas.

Desde un punto de vista conceptual la noción de memoria es simple, sin embargo no existe otro componente en los computadores actuales que posea un abanico de tipos diferentes tan amplio. En general, es deseable que el procesador (CPU) tenga un acceso inmediato e ininterrumpido a cualquier información almacenada en la memoria, de tal manera que la CPU funcione a su velocidad máxima.

Los circuitos de memoria que trabajan a velocidades comparables a las de la CPU son muy costosos. Por este motivo la información en un computador suele estar distribuida sobre dispositivos de memoria basados en diferentes tecnologías, con costes y rendimientos muy diversos y con una organización adaptada a la función específica que desarrollan dentro del computador. Se puede decir que no hay una tecnología óptima para satisfacer todos los requisitos a la vez.

Como consecuencia de esto, un computador contiene una *jerarquía de subsistemas de memoria*, unos internos al propio computador (directamente accesibles desde la CPU) y otros externos (accesibles desde la CPU a través de un módulo de E/S).

En este tema se estudian, en primer lugar, los conceptos básicos y las características de los diferentes tipos de memoria. En segundo lugar se analiza el compromiso que establecen los parámetros de capacidad, velocidad y coste en la jerarquía de memorias y cómo el principio de localidad tanto espacial como temporal permite organizar los datos de una forma eficaz en los diferentes niveles de memoria.

A continuación se estudian de forma más concreta las memorias de tipo semiconductor que utilizan como componente básico los circuitos integrados de memoria, para pasar después a analizar su utilización como componentes del computador. El siguiente apartado se centra en un elemento esencial de todos los computadores actuales: la memoria caché. El tema finaliza estudiando otras arquitecturas de sistemas de memoria como son las memorias de carácter asociativo, la utilización de memoria compartida, el concepto de memorias de tipo pila y una introducción a los discos magnéticos que son uno de los principales sistemas de almacenamiento secundario en los computadores de hoy día.

Sección 2-1

Definiciones y conceptos básicos

El objetivo de cualquier diseño de una unidad de memoria es proporcionar la capacidad de almacenamiento necesaria a un coste razonable y con un nivel de velocidad aceptable.

2.1.1 Localización

Uno de los aspectos más destacables de la memoria de un computador es su *localización* y desde este punto de vista se pueden clasificar en tres grandes grupos:

a) *Memoria interna del procesador*: Esta memoria está constituida, normalmente, por un pequeño conjunto de registros de alta velocidad que son internos a la CPU (ver Figura 1.3). Estos registros los utiliza la CPU como elementos de almacenamiento temporal de instrucciones y de datos, y su estudio será el objeto de temas posteriores.

b) *Memoria principal* (llamada también memoria central o memoria primaria): Es una memoria, relativamente grande y rápida, utilizada para el almacenamiento de las instrucciones de los programas (que están en ejecución en el computador) y sus correspondientes datos. Se caracteriza por el hecho de que se puede acceder a cualquier información almacenada en ella mediante instrucciones del repertorio de la máquina. La tecnología más empleada es la de los circuitos integrados (CI) tipo semiconductor. En los primeros años se utilizaban también memorias principales de soporte magnético (núcleos de ferritas).

c) *Memoria externa o secundaria* (llamada también memoria auxiliar): Son dispositivos periféricos de almacenamiento accesibles a la CPU a través de controladores de E/S. En los dispositivos externos se distingue entre controlador y soporte de la información. El controlador se encarga de dirigir los componentes mecánicos y electrónicos del dispositivo, de esta forma se libera a la CPU de gran parte de la gestión de E/S. El soporte es el medio físico (disco magnético u óptico, cinta magnética, CD-ROM, cinta de papel, tarjeta perforada, etc.) donde se almacena la información. Esta separación facilita el intercambio de información entre dispositivos de almacenamiento que tienen un mismo tipo de controlador. La memoria secundaria suele ser varios órdenes de magnitud más grande y más lenta que la memoria central. Se utiliza fundamentalmente para el almacenamiento de programas y conjuntos de datos que no están siendo utilizados continuamente por la CPU (por ejemplo un compilador de Pascal, programas fuentes, etc.). Para que la CPU pueda acceder a la información contenida en la memoria secundaria es preciso que previamente se cargue esta información, por medio de programas especializados (por ejemplo cargadores), en la memoria principal. Las memorias secundarias que se emplean normalmente son, entre otras, los discos y las cintas magnéticas.

2.1.2 Capacidad

Otra característica importante de la memoria es su *capacidad*, es decir, la cantidad de información que puede almacenar. Ésta se mide en palabras. Así por ejemplo, la capacidad de los registros de la CPU es de una palabra y la de la memoria principal en los computadores actuales es del orden de millones de palabras. Cuando hay grandes necesidades de almacenamiento se recurre a las memorias secundarias.

Es frecuente el uso de dos múltiplos de la unidad de capacidad, éstos son: la *kilopalabra* ($1K$ palabra = 2^{10} palabras = 1.024 palabras) y la *megapalabra* ($1M$ palabra = 2^{20} palabras = 1.048.576 palabras).

La palabra de un elemento de almacenamiento puede constar de varias unidades indivisibles conocidas como *bits* (contracción de las palabras inglesas binary digit). El número de bits por palabra es, la mayoría de las veces, una potencia baja de dos, por ejemplo 3 (en este caso se habla de octetos o bytes). La capacidad de una memoria de p palabras con n bits cada palabra se suele representar como: $p \times n$ bits.

2.1.3 Unidad de transferencia

Un tema relacionado con la capacidad de la memoria es el de la *unidad de transferencia*. En una memoria interna la unidad de transferencia es igual al número de líneas de datos de entrada y salida al módulo de memoria. Este número a menudo coincide con la longitud de palabra, pero puede no hacerlo. Para clarificar este punto se consideran los tres conceptos siguientes que están conectados entre sí en el caso de la memoria interna:

a) *Palabra*: Es la unidad natural de organización de la memoria. El tamaño de la palabra es generalmente igual al número de bits utilizados para representar un número entero y la longitud de una instrucción.

b) *Unidades direccionables*: En muchos sistemas la unidad direccionable es la palabra. Sin embargo, otros permiten un direccionamiento a nivel inferior como por ejemplo un byte. En cualquier caso, la relación entre el número n de bits necesarios para especificar una dirección y el número N de unidades direccionables es: $2^n = N$.

c) *Unidad de transferencia*: Para la memoria principal, la unidad de transferencia es el número de bits leídos de o escritos en la memoria simultáneamente y no tiene por qué ser igual a una palabra o a una unidad direccionable. Para las memorias externas, los datos se transfieren con frecuencia en unidades mucho más grandes que una palabra denominadas *bloques* (aunque no simultáneamente).

2.1.4 Método de acceso

En cualquier operación de lectura o escritura es necesario, en primer lugar, localizar la posición del elemento de almacenamiento donde la CPU debe colocar o extraer la información. Una propiedad muy importante de un dispositivo de memoria es el *orden o método de acceso* en que debe ser localizada o accedida la información. Se distinguen cuatro tipos:

a) *Acceso aleatorio*: Se dice que una memoria tiene acceso aleatorio cuando puede accederse a las informaciones almacenadas en ella en cualquier orden, siendo el tiempo de acceso independiente de la posición donde está localizada la información.

Estas memorias se denominan con el acrónimo de RAM (Random Access Memory). De esta forma

cada posición direccionable de una memoria RAM tiene asignada una única dirección de localización. El tiempo de acceso a cualquier posición es constante e independiente de la secuencia de accesos que haya habido anteriormente. En la Figura 2.1 se muestra en forma de esquema el modelo conceptual de una memoria RAM. La memoria principal de un computador es siempre de acceso aleatorio.

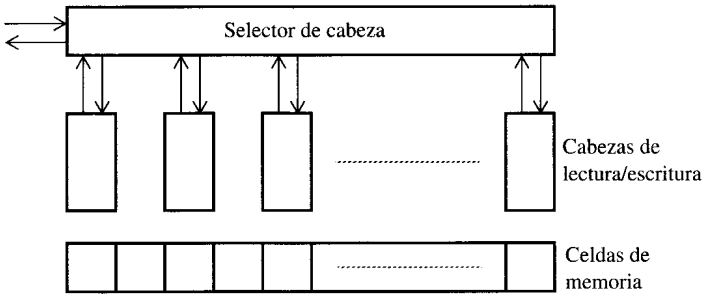


Figura 2.1: Modelo conceptual de una memoria RAM

b) *Acceso secuencial*: Son memorias en las que solamente se puede acceder a la información almacenada en ellas mediante una secuencia lineal.

Por lo tanto, el tiempo necesario para acceder a una información concreta es variable y depende de la posición donde está almacenada. Se denominan con el acrónimo SAM (Sequential Access Memory).

Las unidades de datos elementales de una memoria SAM se llaman *registros*. En la Figura 2.2 se representa el diagrama conceptual de una memoria SAM. Un ejemplo típico de esta clase de memorias son las unidades de cinta magnética.

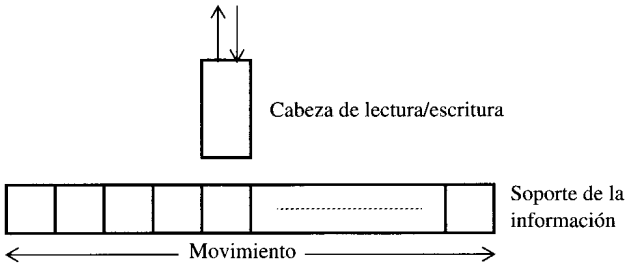


Figura 2.2: Modelo conceptual de una memoria SAM

c) *Acceso directo*: Al igual que en el acceso secuencial, el acceso directo incorpora un mecanismo de lectura-escritura compartido. Sin embargo los bloques o registros individuales tienen una dirección única que se basa en su posición física. El acceso se efectúa mediante un mecanismo de acceso aleatorio para alcanzar la zona más próxima (registro) donde se encuentra la información a la que se desea acceder, y a continuación se realiza una búsqueda de tipo secuencial dentro del

registro. En este caso el tiempo de acceso también es variable. Las unidades de disco son el ejemplo clásico de memoria de acceso directo.

d) *Acceso asociativo*: Son también memorias de acceso aleatorio. Lo que diferencia a este tipo de memorias de las tradicionales, es que en las memorias asociativas no se pregunta por el contenido de una determinada dirección de memoria, ahora se pregunta si existe una posición de memoria que contiene una palabra determinada. Es decir, se selecciona la posición de memoria con la que se desea comunicar, sobre el principio de lo que contiene y no sobre la base de donde está.

A primera vista puede parecer que esta clase de memoria no tiene demasiada utilidad ya que se conoce lo que se quiere buscar. Ahora bien, si se considera que cada palabra de memoria está constituida por dos campos y se pregunta por el contenido de la parte izquierda, la memoria asociativa responde inmediatamente con el contenido de los dos campos (izquierdo y derecho). De esta forma se obtiene una traducción "instantánea" de una zona (izquierda) en la otra (derecha). La ventaja fundamental es que se explora toda la memoria al mismo tiempo y sólo se obtienen las palabras cuyo lado izquierdo es igual al valor buscado. Análogamente se puede preguntar por la zona de la derecha con lo que es igualmente rápida la traducción inversa.

Se puede ver que al igual que la memoria de acceso aleatorio ordinaria, cada posición tiene su propio mecanismo de direccionamiento y el tiempo de recuperación de la información es constante e independiente de la localización y configuración del acceso previo.

2.1.5 Tipos físicos

El sistema de memoria de un computador utiliza diferentes *tipos físicos*. Los tres tipos más usados en la actualidad son:

a) *Memorias de semiconductor*, del tipo LSI (Large Scale Integration) o VLSI (Very Large Scale Integration) utilizadas como memoria principal del computador y donde se incluyen un gran número de distintas tecnologías con diferentes características.

b) *Memorias magnéticas*, utilizadas como memorias secundarias (discos, cintas,...).

c) *Memorias ópticas*, utilizadas también como memorias secundarias.

d) *Memorias magneto-ópticas*, utilizadas también como memorias secundarias.

2.1.6 Características físicas

Resultan particularmente interesantes algunas *características físicas* del almacenamiento de datos. Entre las más sobresalientes están:

a) *Alterabilidad*. Esta propiedad hace referencia a la posibilidad de alterar el contenido de una memoria. Así por ejemplo, una tarjeta perforada puede considerarse como un elemento de almacenamiento de información no alterable. Las memorias cuyo contenido no puede ser modificado se denominan *memorias de sólo lectura* o *memorias ROM* (Read Only Memory). Las diferentes tecnologías proporcionan diversos tipos de memorias ROM de acceso aleatorio, utilizadas fundamentalmente para el almacenamiento de programas en sistemas dedicados a una aplicación específica. Las memorias sobre las que se pueden realizar indistintamente operaciones de lectura o escritura se conocen como *memorias de lectura-escritura* o *memorias RWM* (Read

Write Memory). En la terminología de estructura de computadores es habitual utilizar las siglas RAM para referirse a una memoria de acceso aleatorio y de lectura-escritura.

Algunos tipos de memorias ROM son programables por el usuario, en cuyo caso se las denomina mediante las siglas PROM (Programmable ROM). Otras memorias PROM pueden ser borradas y vueltas a programar con otros contenidos, estas memorias se denominan EPROM (Erasable PROM). El contenido de una memoria EPROM puede borrarse eléctricamente en cuyo caso se conoce como memoria EEPROM (Electrically EPROM), o bien haciendo incidir rayos ultravioletas sobre el circuito de memoria (UVEPROM).

Conviene señalar que el tiempo necesario para escribir una palabra de información en una PROM es muy superior al tiempo necesario para leerla y se requiere de un dispositivo de borrado y grabación específico, lo que obliga a retirarla del sistema para reprogramarla y volverla a instalar después. Este hecho marca la diferencia fundamental con respecto a la ya mencionada memoria RAM.

b) *Permanencia de la información.* Existen tres características de los dispositivos de memoria que pueden redundar en la destrucción de la información que almacenan:

1) Lectura destructiva. Hay memorias, como las de núcleos de ferritas, en las que una operación de lectura ocasiona la destrucción de la información leída. Para evitar la pérdida de información en las *memorias de lectura destructiva* o *memorias DRO* (Destructive Readout), hay que volver a reescribirlas inmediatamente después de haberlas leído. A las memorias de lectura no destructiva se las designa por las siglas NDRO (No Destructive Readout).

2) Volatilidad. Esta característica hace referencia a la posible destrucción de la información almacenada en un cierto dispositivo de memoria cuando se produce un corte en el suministro eléctrico. Las memorias que pierden su información al cesar la tensión de alimentación se denominan *volátiles*, mientras que las que conservan la información se llaman *no volátiles.* Las memorias RAM de semiconductores son volátiles, mientras que las memorias RAM de ferritas y las memorias ROM son no volátiles. Por este motivo, las memorias ROM se utilizan como dispositivo de almacenamiento para los programas de máquinas controladas por computador que, de manera eventual, se pueden desconectar de la fuente de alimentación. De forma análoga, todos los computadores modernos incluyen una pequeña memoria ROM que les permite reinicializar el sistema, cargar el correspondiente sistema operativo y transferirle a éste el control.

3) Almacenamiento estático/dinámico. Una memoria es *estática* si la información que contiene no varía con el tiempo; en el caso contrario, la información almacenada se va perdiendo con el tiempo por lo que hay que refrescarla y se habla en este caso de *memoria dinámica.* El refresco es una dificultad inherente al diseño con memorias dinámicas. Sin embargo, las memorias dinámicas suelen tener una mayor densidad de almacenamiento y un menor consumo de energía que sus equivalentes estáticas por requerir menos elementos electrónicos.

2.1.7 Velocidad

Desde el punto de vista de usuario *velocidad* y *capacidad* son las características fundamentales de la memoria. Para medir el rendimiento en velocidad de una memoria se utilizan los tres parámetros siguientes:

a) *Tiempo de acceso* (t_A). En el caso de lectura de memoria se define como el tiempo medio necesario

para leer una cantidad fija de información, por ejemplo una palabra. Para la escritura en memoria se define de forma análoga y en la mayor parte de los dispositivos es igual al tiempo de acceso para lectura. En otros términos: el t_A de lectura representa el tiempo que ha de pasar desde que se solicita una información a un dispositivo de memoria hasta que está disponible. En memorias de acceso no aleatorio t_A es el tiempo empleado en colocar en su posición la cabeza de lectura-escritura. La Tabla 2.1 muestra valores de algunas clases de memorias.

Tipo de memoria	t_A
Semiconductor bipolar	10^{-8} seg
Semiconductor MOS	10^{-7} seg
Ferritas	10^{-6} seg
Discos magnéticos	10^{-3} seg
Cintas magnéticas	10^{-2} seg

Tabla 2.1: Valores típicos de t_A

b) *Tiempo de ciclo de memoria* (t_C). En las memorias DRO (de lectura destructiva) no es posible iniciar una segunda lectura inmediatamente después de terminada la primera, hay que efectuar antes la restauración de la información leída. Esto significa que el intervalo de tiempo mínimo entre dos lecturas consecutivas puede ser mayor que t_A. Este intervalo se denomina tiempo de ciclo de memoria (ver Figura 2.3).

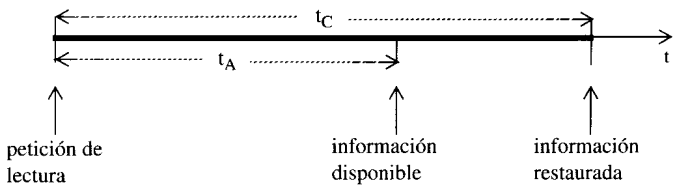


Figura 2.3: Relación entre t_A y t_C

c) *Velocidad de transferencia o frecuencia de acceso* (f_A). En un dispositivo de memoria la velocidad de transferencia se mide como el número de palabras/segundo que pueden ser accedidas. En el caso de memorias de acceso aleatorio f_A es:

$$f_A = \frac{1}{t_C}$$

Para memorias de acceso no aleatorio, se verifica la siguiente relación:

$$t_n = t_A + \frac{n}{P}$$

donde:

t_n = Tiempo medio para leer o escribir n bits

t_A = Tiempo de acceso medio

n = Número de bits que se están transmitiendo

P = Velocidad de transferencia (en bits/seg)

2.1.8 Organización

Se entiende por *organización* la disposición física de los bits para formar palabras. En las memorias de acceso aleatorio la organización es un tema clave de diseño y se estudia posteriormente con mayor extensión.

2.1.9 Resumen de características y propiedades de la memoria

En la Tabla 2.2 se dan las propiedades más importantes de los circuitos integrados de memoria.

Característica	ROM	RAM estática	RAM dinámica
Alterable	No	Si	Si
Capacidad	Muy alta	Alta	Muy alta
Tiempo de acceso	Muy bajo	Muy bajo	Bajo
Refresco	No	No	Si
Volátil	No	Si	Si

Tabla 2.2: Características de los circuitos integrados de memoria

Las características más importantes de los tipos de memoria que se han ido exponiendo en esta sección se resumen en la Tabla 2.3.

Localización	Memoria interna de la CPU Memoria principal Memoria secundaria	**Tipo físico**	Memoria de semiconductor Memoria magnética Memoria óptica
Capacidad	Número de palabras Tamaño de la palabra	**Características Físicas**	Alterabilidad (ROM, RAM) Permanencia de la información - DRO/NDRO - Volátil/No volátil - Estática/Dinámica
Unidad de transferencia	Palabra Bloque	**Velocidad**	Tiempo de acceso t_A Tiempo de ciclo t_C Frecuencia de acceso f_A
Método de acceso	Acceso aleatorio Acceso secuencial Acceso directo Acceso asociativo	**Organización**	2D 21/2D

Tabla 2.3: Características básicas de las memorias

Sección 2-2

Jerarquía de memorias

La unidad de memoria de un computador se puede ver como una jerarquía de componentes. Desde el punto de vista del diseñador, esta jerarquía impone una serie de ligaduras que se pueden resumir en los tres parámetros siguientes (ver Figura 2.4):

- Capacidad de la memoria
- Velocidad
- Coste

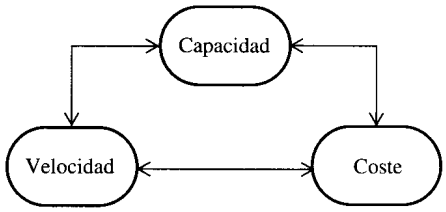


Figura 2.4: Parámetros que hay que considerar en una jerarquía de memorias

La *capacidad* que se requiere en una memoria es siempre un tema abierto que depende de las necesidades de cada usuario y de las aplicaciones que utiliza. Si la capacidad se aumenta es seguro que finalmente se desarrollarán aplicaciones que la utilizan.

La cuestión de la *velocidad* es más fácil de comprender. Para conseguir un rendimiento óptimo del computador la memoria debe ser capaz de seguir el ritmo impuesto por la CPU. Es decir, si la CPU está ejecutando un programa no debería esperar por las instrucciones y los operandos, cuando éstos se le transfieran desde la memoria.

En un computador el *coste* de la memoria suele estar en consonancia con el de las otras unidades del sistema. Cada elemento de almacenamiento diferente tiene un coste asociado. Esta característica puede venir dada en términos absolutos referida al coste del dispositivo de almacenamiento en sí, o en términos relativos referida al coste por unidad de información. Por ejemplo, el coste de una unidad de disco de gran capacidad puede ser superior al de un circuito integrado de memoria, sin embargo, el coste por bit de información almacenado es siempre menor en el caso del disco.

Los tres parámetros antes mencionados están íntimamente ligados. Existe un compromiso entre coste, capacidad y tiempo de acceso. Así, una mayor velocidad suele ir acompañada de un mayor coste y de una

menor capacidad, mientras que por el contrario, una gran capacidad implica un menor coste por unidad de información y una velocidad moderada. La Tabla 2.4 resume estas características.

Menor tiempo de acceso	$\Rightarrow$	Mayor coste por bit
Mayor capacidad	$\Rightarrow$	Menor coste por bit
Mayor capacidad	$\Rightarrow$	Mayor tiempo de acceso

Tabla 2.4: Relación entre capacidad, tiempo de acceso y coste

Teniendo en cuenta las consideraciones anteriores, resulta obvio el dilema que se le presenta al diseñador. Su objetivo será utilizar aquellas tecnologías de memoria que proporcionan una mayor capacidad, ya que ello implica que se tiene tanto la capacidad que se necesita como un coste bajo por unidad de información. No obstante, para poder cumplir los requisitos impuestos por el diseño se ve obligado a utilizar memorias rápidas (con tiempos de acceso pequeños), de baja capacidad y de alto coste. La solución a este conflicto aparente está en emplear una *jerarquía de memorias* en lugar de utilizar una única tecnología o componente de memoria. La Figura 2.5 muestra una jerarquía típica de memorias.

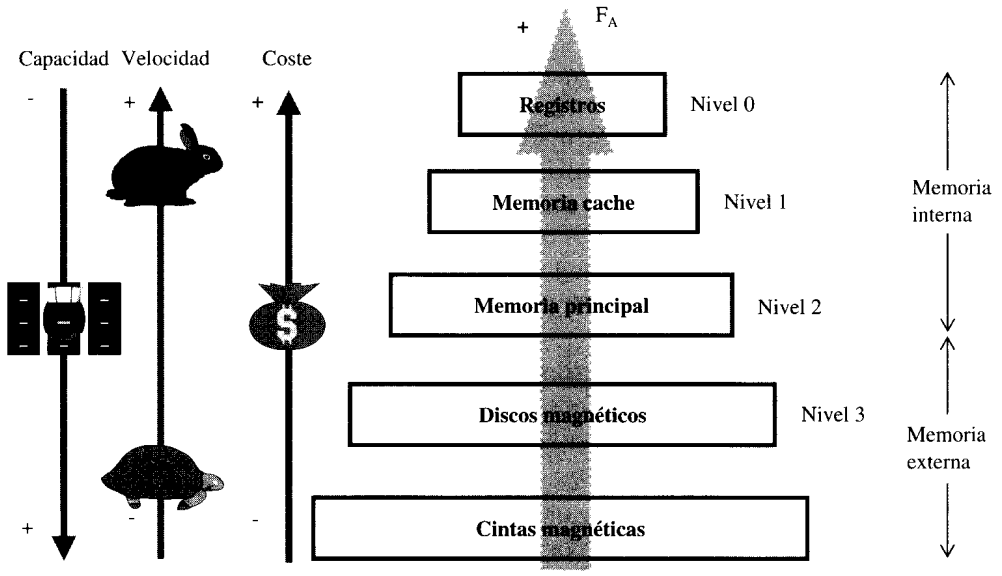


Figura 2.5: Jerarquía de memorias

Cuando se va hacia los niveles inferiores de la jerarquía ocurre que:

a) El coste por unidad de información (bit) disminuye.

b) La capacidad aumenta.

c) El tiempo de acceso aumenta.

d) La frecuencia de acceso a la memoria por parte de la CPU disminuye.

Como consecuencia de esto se deduce que las memorias rápidas, de baja capacidad y de alto coste, se complementan con las memorias lentas, de gran capacidad y de bajo coste. La clave del éxito de esta organización jerárquica está en la característica d). Esta idea se verá posteriormente, con más detalle, cuando se estudie el concepto de "memorias cachés". Una explicación intuitiva de por qué ocurre así es la siguiente:

Si se organiza la memoria de acuerdo con las características anteriores a), b) y c) y si tanto los datos como las instrucciones se pueden distribuir a través de esta memoria de acuerdo con la característica d), es decir, situando en los niveles superiores de la jerarquía los que se necesitan más frecuentemente, y en los niveles inferiores los menos utilizados, resulta evidente que este esquema reducirá el coste total, manteniendo al mismo tiempo un nivel dado de prestaciones. El objetivo final de un sistema de jerarquía de memorias se puede resumir en las dos características siguientes:

1) Ofrecer al usuario tanta *capacidad* de memoria como sea posible al menor *coste*.

2) Con una *velocidad* de acceso cercana a la más rápida necesaria.

A continuación se da un ejemplo sencillo que ilustra este punto.

2.2.1 Ejemplo: Sistema con dos niveles de memoria

La Tabla 2.5 representa las características de dos niveles de memoria a los que tiene acceso la CPU.

	Memoria nivel 1	Memoria nivel 2
Tiempo de acceso	1 μseg	10 μseg
Capacidad	10^3 palabras	10^5 palabras

Tabla 2.5: Características de dos niveles de memoria

Se supone que si la CPU tiene que acceder a una palabra que está en la memoria del nivel 1 lo hace de forma directa, por el contrario si la palabra está en la memoria del nivel 2, se transfiere primero al nivel 1 y desde allí a la CPU. Por simplicidad, no se toma en consideración el tiempo que necesita la CPU para determinar en que nivel está la palabra. El tiempo de acceso medio total (t_A) empleado por la CPU para acceder a una palabra se puede expresar por la siguiente relación:

$$t_A = t_{A1} + t_{A2} - \frac{Tt_{A2}}{100}$$

donde:

t_{A1} = tiempo de acceso a la memoria del nivel 1

t_{A2} = tiempo de acceso a la memoria del nivel 2

T = porcentaje del tiempo total en el que la palabra a la que desea acceder la CPU se encuentra en la memoria del nivel 1

La Figura 2.6 muestra la representación de t_A en función de T. Se ve que cuando $T = 0 \Rightarrow t_A = t_{A1} + t_{A2}$ (todas las palabras buscadas por la CPU están en la memoria del nivel 2), y si $T = 100\ \% \Rightarrow t_A = t_{A1}$ (todas las palabras buscadas por la CPU están en la memoria del nivel 1).

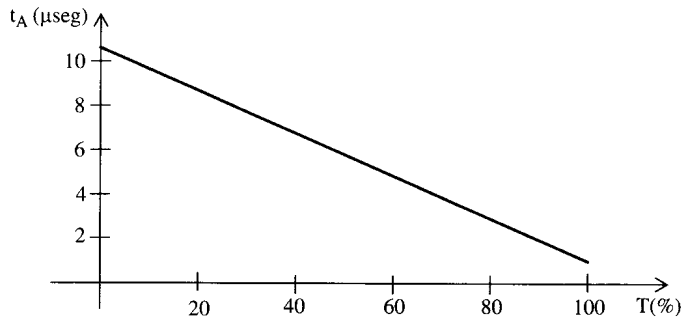


Figura 2.6: Rendimiento de una memoria con dos niveles

En este ejemplo se aprecia que la estrategia de la jerarquía de memorias funciona adecuadamente siempre que se verifiquen las condiciones a), b), c) y d). ♦

La base para la validez de la condición d) es un principio conocido como *principio de localidad.* Esta localidad se puede manifestar de dos formas distintas:

a) *Localidad temporal.* Refleja la tendencia a reutilizar tanto datos como instrucciones a los que se ha accedido recientemente. En los bucles se ejecutan varias veces las mismas instrucciones y las mismas variables se suelen referenciar en múltiples ocasiones.

b) *Localidad espacial.* Refleja la tendencia a referenciar tanto datos como instrucciones que están próximos en su ubicación física en la memoria al encontrarse agrupados en posiciones contiguas. Cuando se está dentro de un bucle o de un procedimiento, hay repetidas referencias a un pequeño conjunto de instrucciones a las que se accede normalmente de forma secuencial. Análogamente, las operaciones sobre ciertas estructuras de datos (listas, arrays, etc.) implican accesos a conjuntos de datos que están próximos entre sí en la memoria. A largo plazo estas agrupaciones cambian, pero a corto plazo la CPU trabaja fundamentalmente con una parte fija de la memoria.

Por este motivo, es posible organizar los datos en los diferentes niveles de memoria, de forma que la frecuencia de acceso sea mucho menor cuanto más abajo se esté en la jerarquía (ver Figura 2.7).

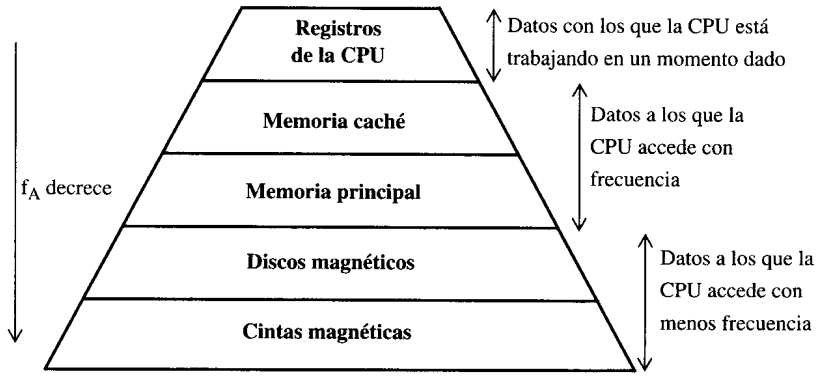


Figura 2.7: Organización de los datos según su f_A

En el caso del ejemplo anterior se puede suponer que la memoria del nivel 2 contiene el programa y los datos, y que las agrupaciones (de datos o instrucciones) que están operativas en un momento determinado se colocan temporalmente en la memoria del nivel 1. De vez en cuando, una de las agrupaciones del nivel 1 se tendrá que mover al nivel 2 y liberar espacio para una nueva agrupación. No obstante, en promedio, la mayoría de las referencias que efectúa la CPU serán a datos e instrucciones contenidos en el nivel 1.

El resto de este tema se dedica a estudiar los niveles superiores de la jerarquía (memoria interna) que aparecen en la Figura 2.5.

Sección 2-3

Memorias de semiconductor

En los primeros computadores la forma más común de memoria principal de acceso aleatorio utilizaba una matriz de pequeñas bobinas ferromagnéticas conocidas como *núcleos de ferrita*. Por ello a la memoria principal se la denominaba a menudo *memoria de núcleo*, término que persiste aún hoy día. La aparición de la microelectrónica, con las ventajas inherentes de los dispositivos electrónicos de estado sólido, han hecho desaparecer prácticamente de la escena a las memorias de núcleo magnético.

Los semiconductores ya se empleaban desde los años 60 para la fabricación de los circuitos internos de la CPU. Sin embargo, solamente después de la aparición, a finales de esa década, de las técnicas microelectrónicas de producción de circuitos integrados se hizo posible la construcción de memorias de semiconductor de alta capacidad a un coste razonable, utilizando como componentes básicos los Circuitos Integrados de Memoria (CIM).

2.3.1 Características generales de un CIM

Un CIM está organizado internamente como una matriz de $N \times m$ celdas elementales, en las que se pueden almacenar N palabras de m bits tal como se indica en la Figura 2.8.

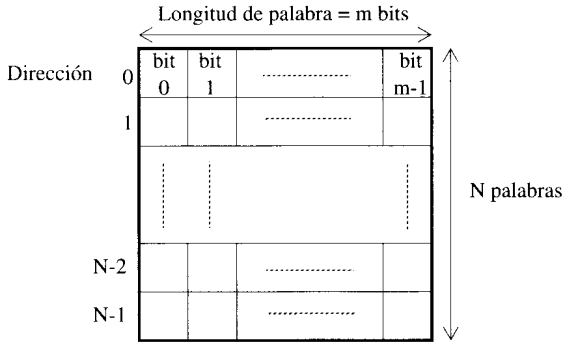


Figura 2.8: Esquema de la memoria de un CIM

A cada palabra almacenada en el CIM se le asigna una única dirección. Dándole al CIM la dirección adecuada se puede extraer de él cualquier palabra que se desee, con independencia de su localización física dentro del mismo. Un CIM con capacidad para N palabras, de m bits cada una, tiene la estructura que se muestra en el diagrama de bloques de la Figura 2.9.

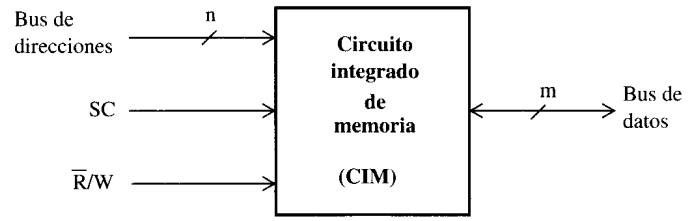


Figura 2.9: Estructura funcional de un CIM

El número de líneas n del bus de dirección es tal que $2^n = N$, siendo N el número de palabras del CIM. El número de líneas m del bus de datos se corresponde con la longitud de palabra (o un divisor de ella). *SC* (Selección de Circuito, del inglés "Chip Select") es una línea general de habilitación que debe estar activada para que la memoria responda a cualquier petición de lectura o escritura. La línea $\overline{R}/W$ (Read/Write) es una entrada de selección que solamente estará presente en los circuitos de memoria de lectura/escritura (memorias RAM), y su misión consiste en seleccionar en un momento dado si el CIM ha de realizar una operación de lectura o una operación de escritura. En la Tabla 2.6 se resumen las operaciones que realiza un CIM en función del valor de sus entradas de control.

SC	R/W	Operación de memoria
0	×	Ninguna
1	0	Leer de la palabra seleccionada
1	1	Escribir en la palabra seleccionada

Tabla 2.6: Entradas de control de un CIM

En las Figuras 2.10 y 2.11 se incluyen los cronogramas representativos de los ciclos de lectura y escritura de una memoria RAM. Estos cronogramas son datos que ofrecen los fabricantes de memorias y ponen de manifiesto las acciones que debe realizar la CPU para leer o escribir en un determinado circuito.

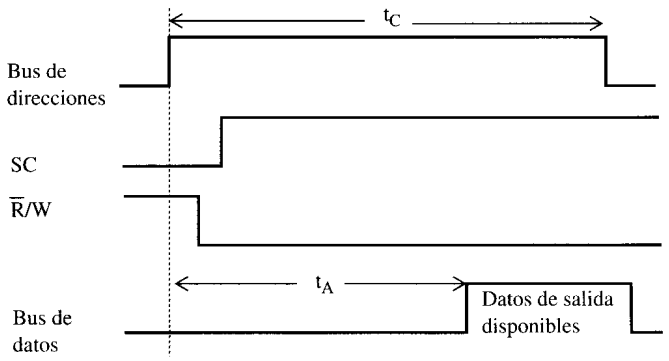


Figura 2.10: Ciclo de lectura de un CIM

Un ciclo de lectura o de escritura comienza colocando una nueva dirección en el bus de direcciones. La línea *SC* de selección del circuito debe activarse (*SC* = 1) después de que la dirección se encuentre disponible. *SC* se debe mantener activa durante toda la secuencia de ciclos de memoria en los que se utilice el mismo circuito. En el caso de una operación de lectura la señal $\overline{R}/W$ debe ponerse al nivel lógico 0 y puede permanecer en este valor durante toda la secuencia de ciclos de lectura. Los datos a leer están disponibles en el bus de datos de salida t_A (tiempo de acceso) segundos después de transmitir la dirección, tal como se muestra en la Figura 2.10. En una operación de escritura (ver Figura 2.11) los datos a escribir se envían al bus de datos de entrada aproximadamente al mismo tiempo que se coloca la dirección en el bus de direcciones.

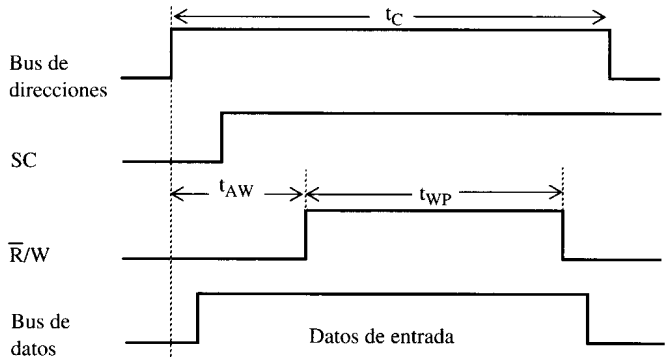


Figura 2.11: Ciclo de escritura de un CIM

Transcurrido t_{AW} segundos (conocido como el tiempo de fijación de la dirección) la línea $\overline{R}/W$ debe pasar de 0 a 1 para iniciar el proceso de escritura, que debe durar como mínimo un tiempo t_{WP} (conocido como anchura del pulso de escritura), para garantizar que los datos han pasado ya a la memoria RAM. Durante este tiempo no deben cambiar los datos de entrada. El tiempo de ciclo de escritura t_W se define como: $t_W = t_{AW} + t_{WP}$.

2.3.2 Ejemplo: Cálculo del número de ciclos de reloj en los accesos a memoria

Una CPU opera con una frecuencia de reloj de 50 MHz, lo que da un período para cada pulso de reloj de 20 ns (1 ns = 10^{-9} s). Se supone que la CPU se comunica con su memoria con un tiempo de acceso de t_A = 65 ns y un tiempo de ciclo de escritura de t_W =75 ns. El número de pulsos de reloj que se requieren para hacer una petición a memoria es el valor entero mayor o igual al máximo del tiempo de acceso y del tiempo de ciclo de escritura, dividido entre el período del reloj. Como el período del reloj de la CPU es de 20 ns y el máximo de t_A y t_W es 75 ns, serán necesarios dedicar al menos cuatro ciclos de reloj a cada petición de memoria. ♦

2.3.3 Estructura de la celda básica de memoria

El elemento básico de un CIM es su celda de memoria que permite almacenar un bit de información. Aunque se utilizan diferentes tecnologías electrónicas, todas las celdas de memoria de tipo semiconductor comparten ciertas propiedades:

- La celda de memoria presenta dos estados que se pueden usar para representar el 1 y el 0 binario.

- La celda de memoria se puede escribir (al menos una vez) para fijar su estado.
- La celda de memoria se puede leer para conocer el valor de su estado.

En el caso de las memorias ROM la celda básica de almacenamiento es un simple conmutador de tipo semiconductor que está permanentemente conectado o desconectado, generando como consecuencia un bit de datos constante, 1 ó 0, cada vez que se selecciona.

Las celdas en una memoria RAM son inherentemente más complejas que en una memoria ROM, ya que se puede leer y escribir y todo ello durante las condiciones normales de funcionamiento. En la Figura 2.12a se representa una celda de tipo RAM realizada con puertas lógicas y un biestable *RS*. La celda se selecciona con la señal *Selección* = 1 cada vez que se quiere leer o actualizar su contenido (un bit). La operación de escritura se distingue de la de lectura porque la señal $\overline{Lectura/Escritura}$ = 1. En el caso de una operación de lectura (*Selección* = 1 y $\overline{Lectura/Escritura}$ = 0), las entradas al elemento biestable son: $S = 0$ y $R = 0$, con lo cual no modifica su estado; es decir, la *Salida* de la celda toma el valor del bit que estuviera almacenado en ella.

Cuando la operación es de escritura (*Selección* = 1 y $\overline{Lectura/Escritura}$ = 1), las entradas al biestable son S = *Entrada* y $R = \overline{Entrada}$ y siendo *Entrada* el bit que se desea almacenar en la celda. Si *Entrada* = 0 $\Rightarrow$ $S = 0$ y $R = 1$, esto supone una acción de puesta a 0 para el biestable ($Q = 0$) por lo que *Salida* = *Entrada* = 0. Si *Entrada* = 1 $\Rightarrow$ $S = 1$ y $R = 0$, que representa una acción de puesta a 1 para el biestable ($Q = 1$), por lo que *Salida* = *Entrada* = 1. En la Figura 2.12b se muestra un esquema de la celda básica que se va a utilizar como bloque elemental al explicar las diferentes alternativas de organización de las memorias RAM.

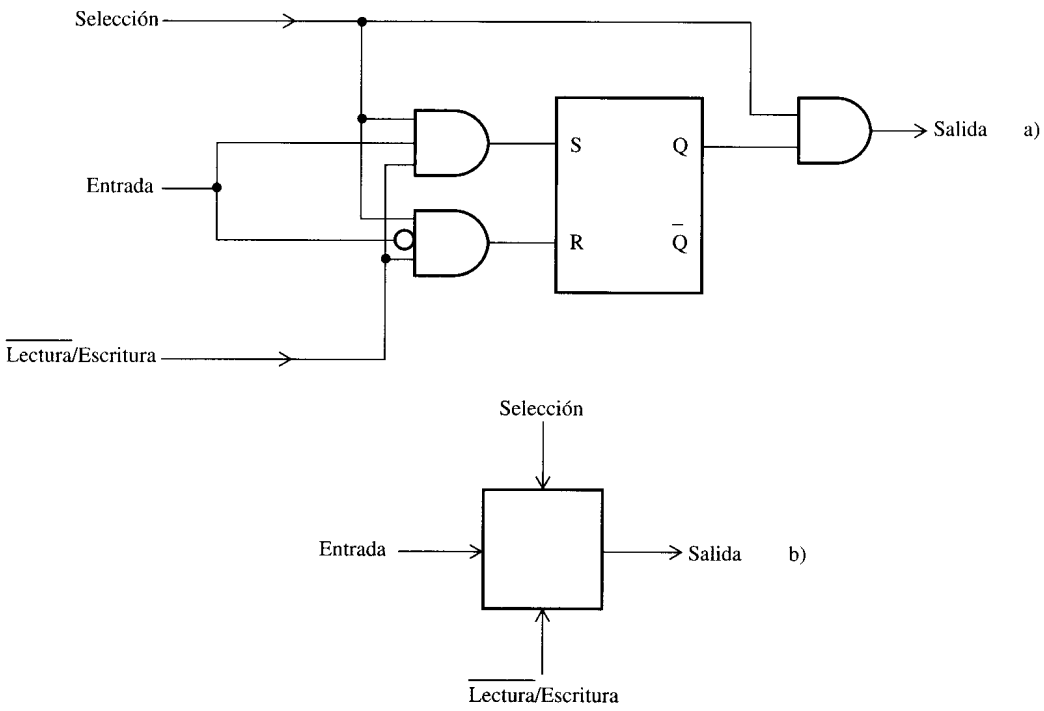


Figura 2.12: a) Celda básica de memoria RAM b) Esquema de la celda básica de memoria

El funcionamiento descrito anteriormente de la celda de memoria RAM se resume en la Tabla 2.7

Selección	Escritura	Contenido	Salida
0	x	Contenido	0
1	0	Contenido	Contenido
1	1	Entrada	Entrada

Tabla 2.7: Tabla de verdad de la celda básica RAM

La celda RAM que se acaba de presentar es de tipo *estática* porque, salvo cuando se escriben nuevos datos en la memoria, los datos almacenados se mantienen sin alteración por un tiempo indefinido. En las memorias RAM *dinámicas* los datos almacenados se desvanecen gradualmente y para que no se pierdan deben ser restaurados, a intervalos de tiempo regulares, mediante un proceso conocido como *refresco*. Para refrescar un dato no es necesario volver a escribirlo y por tanto puede ser realizado por un circuito específico distinto a la CPU. Las principales ventajas de las celdas de tipo RAM dinámicas es su sencillez y una mayor capacidad de almacenamiento que las estáticas.

2.3.4 Organización interna

Cada CIM contiene una matriz de celdas de memoria. Con las tecnologías actuales de fabricación de circuitos integrados es posible encapsular en una única unidad $64K$ bits, $256K$ bits, $1M$ bits, $4M$ bits, $8M$ bits, $16M$ bits, $32M$ bits y $64M$ bits. En el apartado de jerarquía de memorias se mostró que había un compromiso entre velocidad, capacidad y coste. También existe este mismo compromiso cuando se considera la organización de las celdas de memoria y la lógica de control y selección asociada a un CIM.

La Figura 2.9 correspondía a la estructura funcional de un CIM considerado desde un punto de vista externo, ya que pone de manifiesto las señales que el módulo de memoria intercambia con el exterior. Internamente la estructura de un CIM puede variar dependiendo de las ligaduras de diseño que se le impongan. Se suelen utilizar dos organizaciones conocidas como 2D y 21/2D.

Organización 2D

Es el método más simple de organizar una memoria de semiconductor, pero desde el punto de vista de su realización física es el de coste más elevado. La organización 2D se conoce también como organización por palabras u organización lineal.

En este tipo de organización el módulo de memoria se compone de una matriz de celdas básicas (del tipo descrito anteriormente) y de los mecanismos de control y selección que se indican en la Figura 2.13.

Para una memoria RAM de 2^n palabras de m bits cada una, la matriz de celdas está formada por 2^n filas y m columnas. Además del array de celdas el módulo para seleccionar la palabra deseada requiere un decodificador de n entradas y 2^n salidas y para formar la salida m puertas OR de 2^n entradas cada una . En la organización 2D la disposición física de la matriz de celdas es la misma que la disposición lógica (tal como la percibe el procesador) de las palabras de memoria. Es decir no hay diferencia entre el tamaño de la *palabra física* y la *palabra lógica* (m bits). El tamaño de palabra física es el número de celdas en cada fila de la memoria, en cambio el tamaño de palabra lógica está determinado por el tipo de acceso permitido. En la Figura 2.14 se representa una memoria RAM de 4 palabras con 4 bits por palabra.

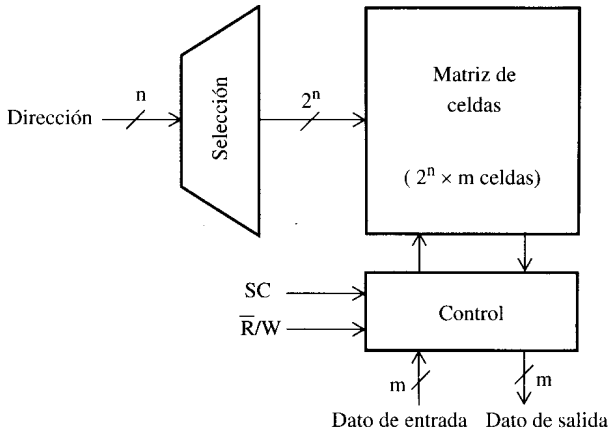


Figura 2.13: Organización 2D de una memoria RAM

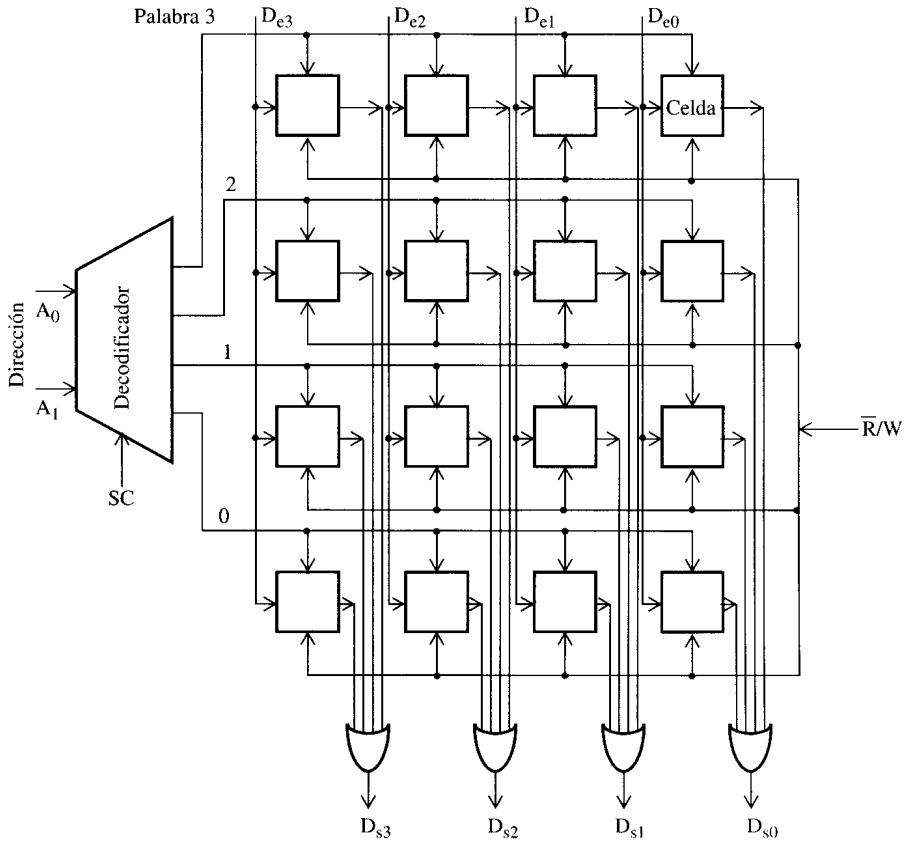


Figura 2.14: Memoria RAM de 4 palabras con 4 bits por palabra

Cada una de las salidas del decodificador está asociada con una palabra de la memoria. Por ejemplo, una entrada al decodificador igual a $A_0 = 0$ y $A_1 = 1$ origina que se active la línea de salida del decodificador número 2 (las líneas se numeran desde 0 a N-1 = 2^n-1). Si la operación es de lectura ($\overline{R}/W = 0$) sólo la palabra de memoria seleccionada vierte los datos que contiene a las líneas de salida de datos (D_{s3}, D_{s2}, D_{s1}, D_{s0}). De forma análoga, en las operaciones de escritura ($\overline{R}/W = 1$) el dato presente en las líneas de entrada de datos (D_{e3}, D_{e2}, D_{e1}, D_{e0}) se introduce en la celda correspondiente de la palabra de memoria seleccionada. La línea de selección de circuito (SC) permite la inhibición total de la actuación del decodificador, independientemente de la configuración de entrada que tenga presente. Esta inhibición impide que el contenido de cualquier palabra de la memoria pase a las líneas de datos y es un factor fundamental, como se verá más adelante, para el diseño de unidades de memoria basadas en la utilización reiterada de un módulo básico.

La estructura 2D es una organización rápida ya que el único retardo que presenta es el asociado con los circuitos de decodificación y con el acceso a los elementos físicos de almacenamiento. En principio y para un número dado M de bits ($M = 2^n \times m$) se podría utilizar la organización 2D para diseñar una memoria con un número arbitario de filas y columnas. Hay no obstante dos factores que limitan la aplicabilidad de esta organización:

1) *Complejidad del decodificador de direcciones.* El decodificador de direcciones requiere al menos una puerta AND por cada una de las filas de la matriz de celdas. En otras palabras, el decodificador de direcciones necesita 2^n puertas AND.

2) *Número de líneas de dirección y datos.* La organización 2D precisa de $n + m$ líneas (n líneas de dirección y m líneas de datos).

En la Tabla 2.8 se muestran los valores de complejidad del decodificador y número de líneas para una memoria de 4K bits. Al examinar esta tabla se observa que cuanto más cuadrada es la memoria menor es el coste del decodificador y mayor el número de líneas que se necesitan.

Tamaño	Complejidad del decodificador	Número de líneas
$2^6 \times 64$	64 puertas AND	6 + 64 = 70
$2^7 \times 32$	128 puertas AND	7 + 32 = 39
$2^8 \times 16$	256 puertas AND	8 + 16 = 24
$2^9 \times 8$	512 puertas AND	9 + 8 = 17
$2^{10} \times 4$	1024 puertas AND	10 + 4 = 14
$2^{11} \times 2$	2048 puertas AND	11 + 2 = 13
$2^{12} \times 1$	4096 puertas AND	12 + 1 = 13

Tabla 2.8: Número de líneas y complejidad del decodificador para una memoria de 4 Kbits con organización 2D

Como en un circuito integrado de memoria, normalmente, el número de palabras (2^n) es mucho mayor que el número de bits por palabras (m), la utilización de una organización 2D dará lugar a matrices de celdas excesivamente largas y estrechas, que no resultan adecuadas para su realización en un circuito integrado. Estas desventajas de la organización 2D se trata de evitar con la organización 2½D.

Organización 21/2D

Con la organización 2D existen dos problemas para la realización de módulos de memoria RAM de gran capacidad:

a) Las m puertas lógicas OR tienen demasiadas entradas (2^n cada una).

b) El decodificador de direcciones es complejo y costoso: Requiere 2^n puertas lógicas AND con n entradas por puerta y 2^n conexiones que van desde el decodificador a la matriz de celdas.

En la organización 2D todos los bits de una palabra dada están en el mismo circuito integrado, sin embargo en la organización 21/2D están localizados en más de un circuito integrado. Por ejemplo, una palabra de 16 bits se podría almacenar en 4 circuitos integrados con 4 bits en cada uno de ellos. El caso más extremo de esta organización, y también el más común, es cuando cada bit de una palabra se encuentra en un circuito integrado distinto.

Para reducir los requisitos mencionados la organización 21/2D emplea un decodificador por coincidencia (por este motivo a la organización 21/2D se la llama también organización por coincidencia). Consiste en utilizar, en lugar de un único decodificador con n entradas y 2^n salidas, dos decodificadores con $n/2$ entradas y $2^{n/2}$ salidas en cada uno de ellos. Uno de los decodificadores realiza la selección sobre el eje X y el otro sobre el eje Y, tal como se muestra en la Figura 2.15. De esta manera se pueden conseguir matrices de celdas de memoria más cuadradas que con la estructura 2D, además de reducir el número de puertas.

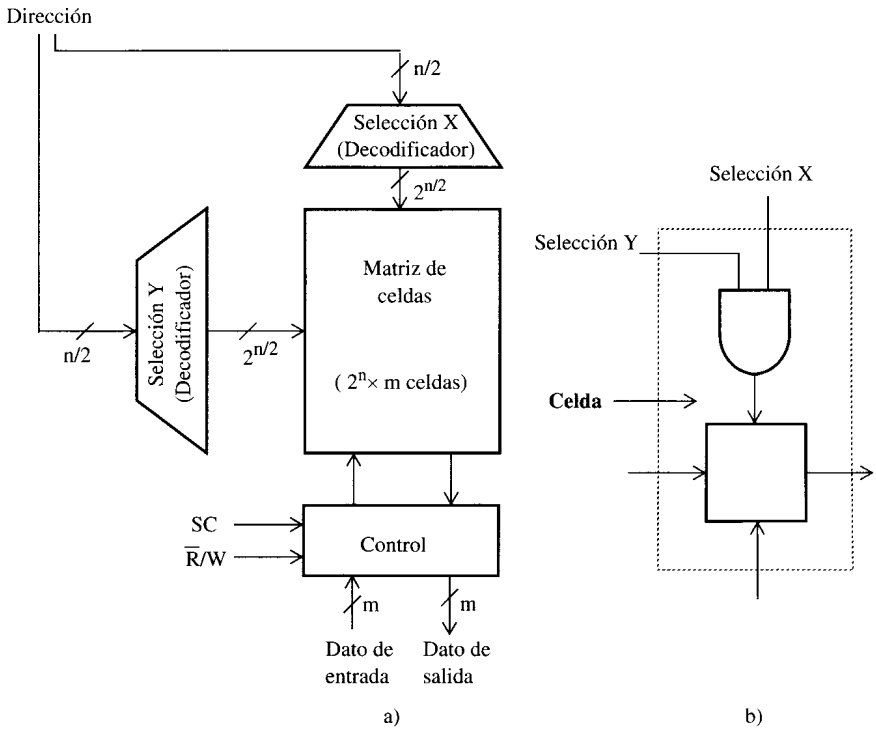


Figura 2.15: a) Memoria RAM con decodificación por coincidencia b) Celda básica de memoria modificada

Esta configuración con dos decodificadores requiere menos puertas lógicas con menor número de entradas por puerta que cuando se utiliza un único decodificador y el número de conexiones que van de los decodificadores a la matriz de celdas de memoria se reduce de 2^n a $2 \times 2^{n/2}$. Sin embargo, este esquema utiliza una celda básica de memoria algo más complicada, ya que incluye una puerta lógica AND necesaria para efectuar la selección por coincidencia.

Para seleccionar el bit de una palabra en particular la dirección de la palabra se divide en dos mitades. La primera parte de la dirección se introduce al decodificador *Y*, que selecciona una fila igual que en la organización 2D. El resto de la dirección va al decodificador *X* conectado a las columnas de la matriz de celdas de memoria. Este segundo decodificador selecciona una columna de la matriz, permitiendo así que sólo se lea o se escriba en una única celda de memoria (la celda que se encuentra en la intersección de la fila y la columna seleccionadas por los dos decodificadores). En la Figura 2.16 se muestra un esquema de selección por coincidencia para el caso de una memoria RAM de 16 palabras de 1 bit cada una (sólo se representan las líneas de dirección). Con el fin de compararla con la organización 2D en la Figura 2.17 se presenta la misma memoria con selección lineal.

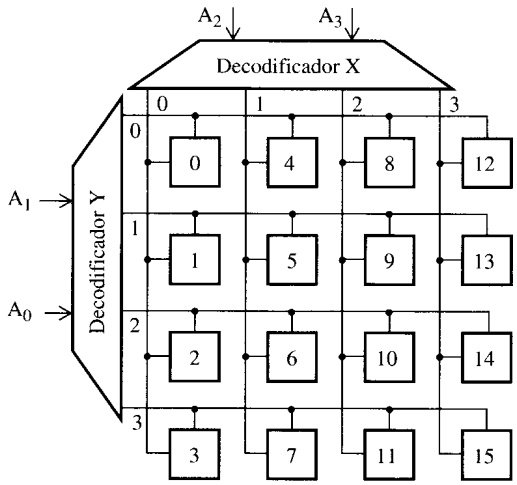


Figura 2.16: Memoria RAM de 16 × 1 con selección por coincidencia

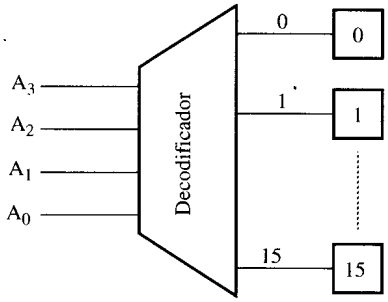


Figura 2.17: Memoria RAM de 16 × 1 con selección lineal

En la organización 21/2D si que se establece una diferencia entre el tamaño de la *palabra física* y la *palabra lógica*. La memoria que se muestra en la Figura 2.18 tiene una palabra física de 64 bits, sin embargo sólo permite el acceso a valores de un bit y por lo tanto tiene una palabra lógica cuyo tamaño es de un bit.

Aunque la organización 21/2D ofrece una solución parcial al problema del número de conexiones externas al ser la matriz de celdas cuadradas, esta solución no es suficiente para circuitos de memoria de gran capacidad. La tecnología actual de fabricación de circuitos integrados permite disponer hasta 64 Mbits en un sólo chip. Con las técnicas que se han analizado, un chip de $4M$ bits por ejemplo requeriría al menos 27 conectores (22 conectores de dirección, 1 conector de datos, 2 conectores de señales de control, 1 conector de alimentación del chip y 1 conector de tierra).

Se puede emplear otra técnica la *multiplexación* para reducir el número de terminales necesarios para suministrar el valor de la dirección al chip. En la multiplexación los mismos conectores transmiten valores distintos en instantes diferentes lo que hace más lentos los tiempos de acceso y ciclo de memoria pero permiten como contrapartida una mayor densidad de celdas de memoria.

El uso combinado de direccionamiento multiplexado y de matrices de celdas cuadradas dan lugar a que la capacidad de la memoria se multiplique por cuatro con cada nueva generación de circuitos de memoria. La inclusión de un terminal adicional dedicado a direccionamiento duplica el número de filas y columnas de manera que el tamaño del circuito se incrementa por un factor de 4. El ritmo de evolución que se ha tenido en cuanto a capacidades de los circuitos de memoria ha sido el de una nueva generación cada tres años: $1K$, $4K$, $16K$, $64K$, $256K$, $1M$, $4M$, $16M$, $64M$, ...

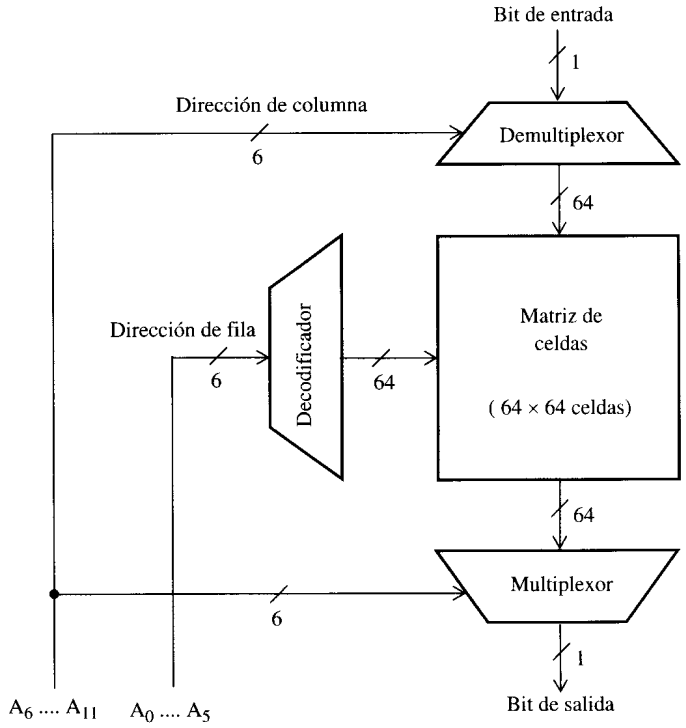


Figura 2.18: Organización 21/2D de una memoria de 4096 x 1 bits

Para realizar una memoria RAM de 2^n palabras × m bits con selección por coincidencia, se repite m veces el plano de 2^n palabras × 1 bit con los decodificadores X e Y compartidos por los diferentes planos. En la Figura 2.19 se muestra un ejemplo de una memoria RAM de 16 × 2 (sólo se representan las líneas de direcciones).

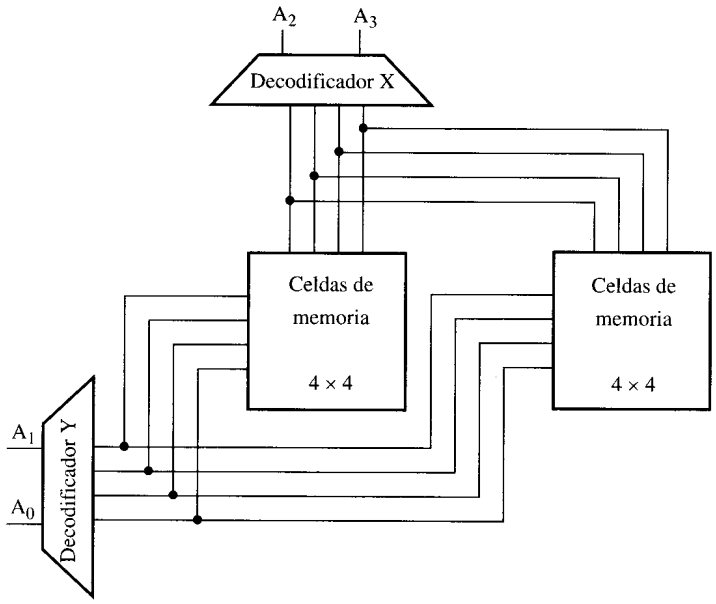


Figura 2.19: Memoria RAM de 16 × 2 con selección por coincidencia

2.3.5 Diseño de bloques de memoria

La ampliación de componentes es una característica fundamental en el diseño y en el caso de las memorias de semiconductor tiene dos objetivos:

a) Incrementar el tamaño de las palabras (número de bits/palabra).

b) Incrementar el número de palabras de la memoria.

Ampliar el tamaño de las palabras de memoria no presenta mayores dificultades. Supóngase que se pretende construir un bloque de memoria RAM de N palabras × km bits (siendo $N = 2^n$), utilizando módulos de memoria de N palabras × m bits (ver Figura 2.20).

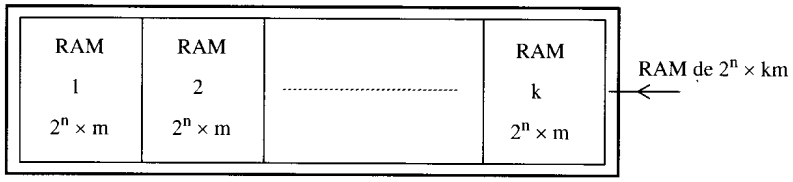


Figura 2.20: Composición de una memoria RAM de 2^n × km bits con k RAM's de 2^n × m bits

Para ello basta con utilizar k módulos e interconectarlos tal como se indica en la Figura 2.21. Es decir, las líneas externas del bus de dirección y las líneas de control son comunes a todos los circuitos, mientras que los k buses de datos se yuxtaponen para formar un único bus de datos de km bits.

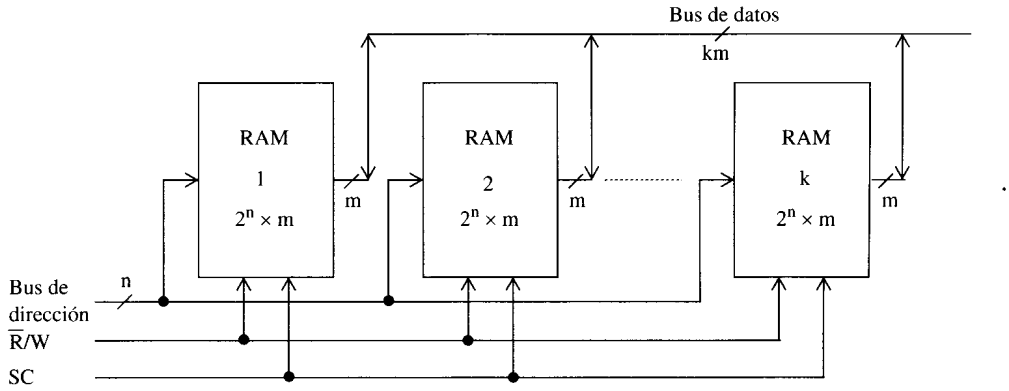


Figura 2.21: Diseño de una memoria RAM de N × km bits con k RAM's de N × m bits

Incrementar el número de palabras supone aumentar la capacidad de direccionamiento y esto conlleva un incremento del número de bits del bus de dirección. En la Figura 2.22 se ilustra cómo se puede construir un bloque de memoria RAM de 2^kN palabras × m bits, con 2^k módulos de memoria de N palabras × m bits.

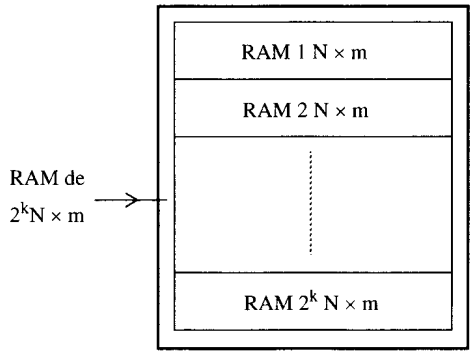


Figura 2.22: Composición de una memoria RAM de 2^kN × m bits con 2^k RAM's de N × m bits

En la Figura 2.23 se muestra el diseño de una memoria RAM con capacidad de $2^kN \times m$ bits utilizando 2^k memorias RAM de $N \times m$ bits. Tal como aparece en la Figura 2.23 esto requiere el uso de un decodificador adicional, ya que el número de líneas del bus de dirección tendrá que incrementarse en k. Las n líneas menos significativas del bus de dirección: A_{n-1},..., A_0 son compartidas por todos los circuitos de memoria. Las restantes k líneas del bus de dirección: A_{n+k-1},...., A_n se conectan a las entradas de un circuito decodificador de k entradas y 2^k salidas. Las salidas de este decodificador sirven de líneas de selección a los 2^k módulos de N palabras × m bits.

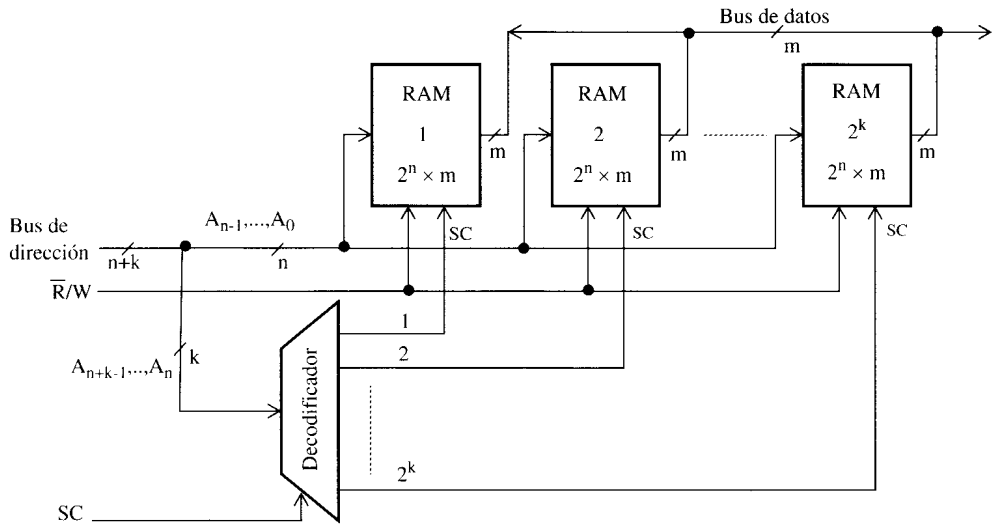


Figura 2.23: Diseño de una memoria RAM de 2^kN × m bits con 2^k RAM's de N × m bits

Ambas técnicas de ampliación pueden combinarse fácilmente para construir bloques de memoria tan grandes como se quiera a partir de otros más pequeños.

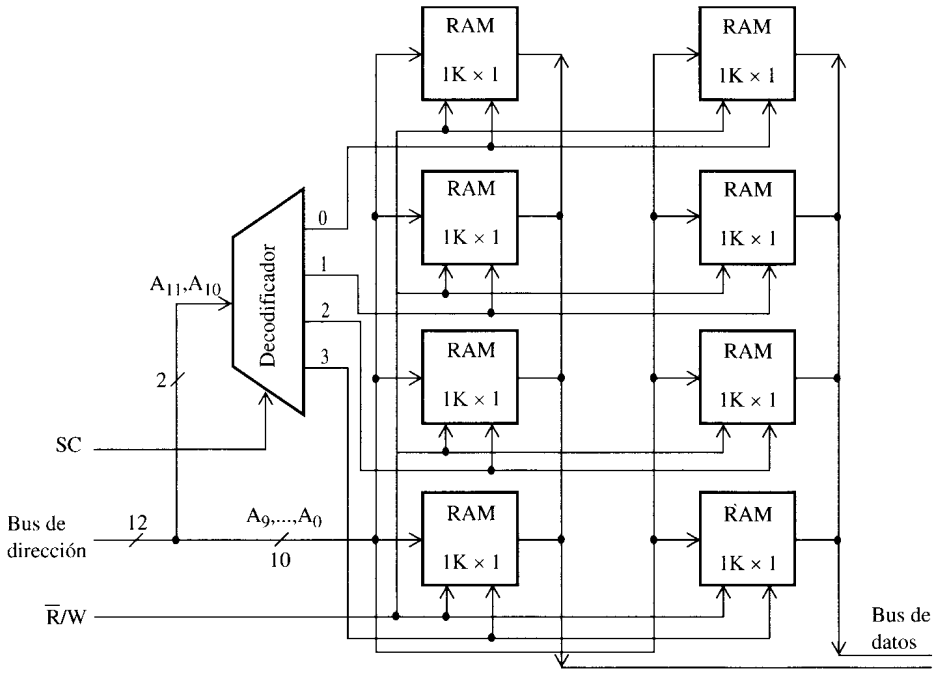


Figura 2.24: RAM de 4096 × 2 bits construida con 8 RAM's de 1024 × 1 bits

La Figura 2.24 representa un bloque de memoria RAM de 4.096 palabras × 2 bits, construido con 8 memorias RAM de 1.024 palabras × 1 bit cada una y un decodificador de 2 a 4 (que permite seleccionar cada fila). Tal como se han conectado los circuitos, las memorias RAM situadas en una misma columna almacenan un bit de cada palabra de memoria.

2.3.6 Conexión de la unidad de memoria al bus del sistema

Una vez examinado el diseño de un bloque de memoria a partir de un determinado tipo de circuito integrado se pasa a considerar, brevemente, la conexión de las líneas de dirección y de datos entre el procesador y la memoria mediante los correspondientes drivers.

El esquema de drivers que se presentó en el apartado 1.4.5 del tema 1 permite realizar la conexión del bloque de memoria con el bus. Esta conexión es diferente según el tipo de líneas del bus de que se trate.

a) *Bus de dirección y bus de control.*

Ambos buses son unidireccionales, es decir, la transferencia de información se realiza en un sólo sentido, desde la CPU a la memoria.

b) *Bus de datos*

Es el caso más complicado ya que el bus de datos es bidireccional y es capaz de transferir información en ambos sentidos, desde la CPU a la memoria (operación de escritura) y desde la memoria a la CPU (operación de lectura). Se pueden distinguir dos situaciones, según que el bloque de memoria disponga de entrada y salida de datos separadas (ver Figura 2.25), o las tenga en común sobre una sola línea bidireccional (ver Figura 2.26). En ambas figuras únicamente se representan las conexiones de las líneas de datos.

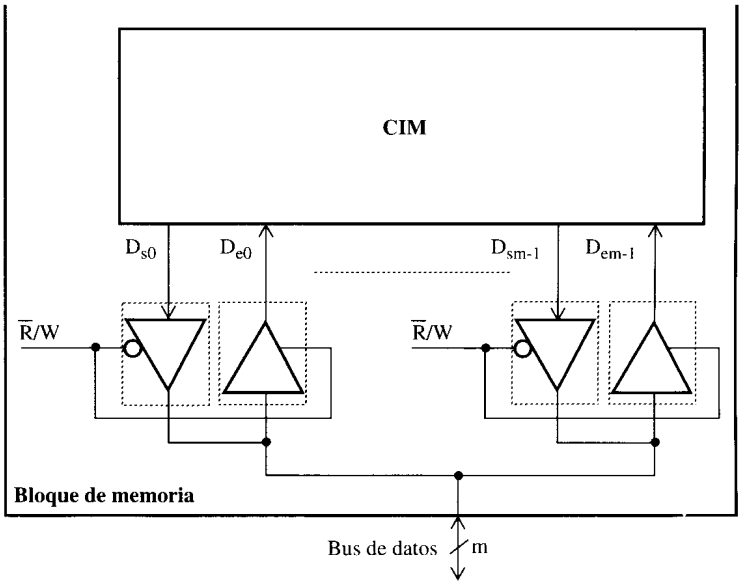


Figura 2.25: E/S de datos por líneas separadas

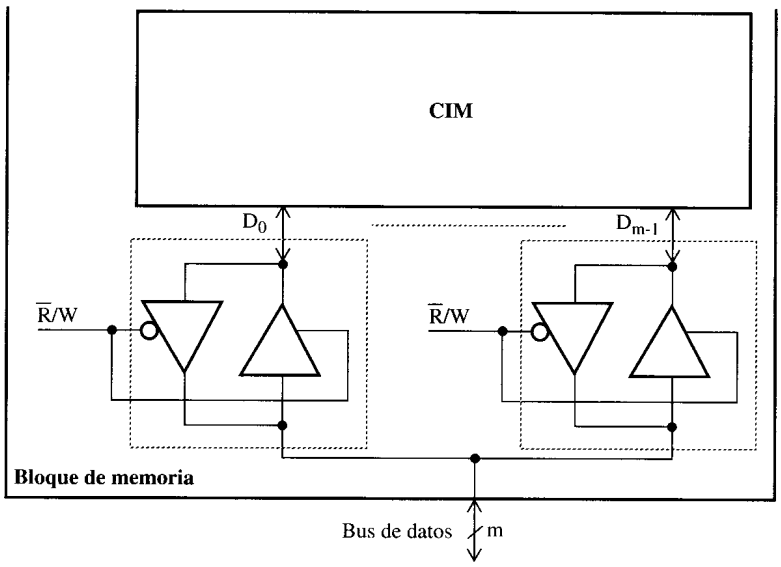


Figura 2.26: E/S de datos por una línea común

2.3.7 Estructura y direccionamiento de la unidad de memoria

En el diseño de la unidad de memoria principal se utiliza un bloque básico, que se dispone en una placa de circuito impreso tal como se muestra en la Figura 2.27 y que se repite tantas veces como sea necesario.

Este enfoque presenta ciertas ventajas desde el punto de vista del diseño global de la memoria:

a) Abaratamiento de los costes, ya que se hace un sólo diseño y se construye una única placa.

b) Modularidad y posibilidad de ampliación de la memoria, insertando nuevas placas en el bus.

c) Facilidad de mantenimiento y reparación simplemente cambiando la placa averiada por una nueva.

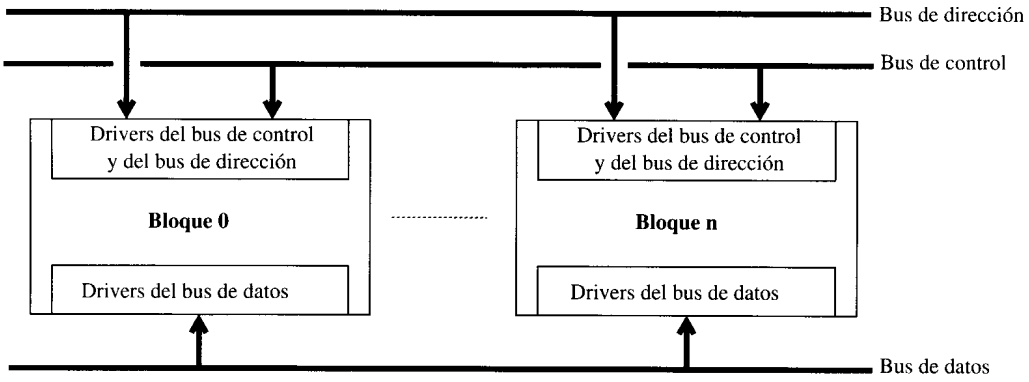


Figura 2.27: Diseño de la unidad de memoria a partir de un bloque básico

Como ejemplo para fijar las ideas, se considera un procesador que dispone de 20 líneas de dirección: A_{19}, A_{18}, A_{17}, ..., A_0. El conjunto de direcciones tendrá (en notación hexadecimal) todos los valores posibles entre 00000 y FFFFF.

Si se dispone de bloques de memoria de $256K$ palabras para construir la unidad de memoria asociada a este procesador, cada bloque de memoria puede representarse tal como se indica en la Figura 2.28.

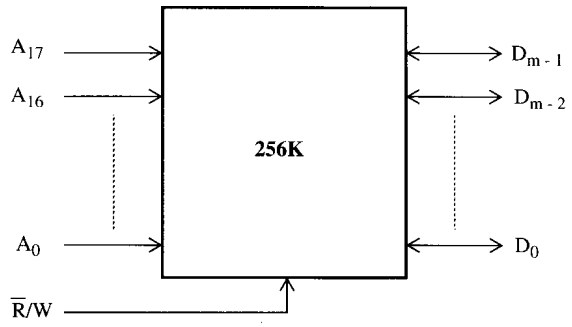


Figura 2.28: Bloque de memoria RAM de 256K

Con este bloque se puede cubrir el mapa de memoria del computador según se muestra en la Figura 2.29.

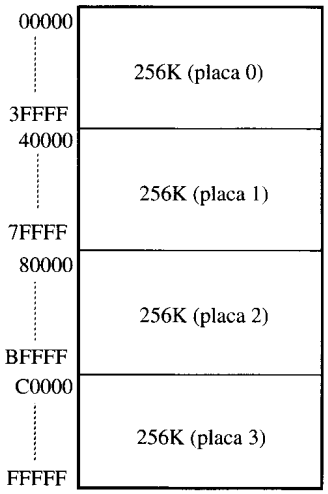


Figura 2.29: Mapa de memoria

A la vista del mapa de memoria es preciso lograr que, para todos los accesos a palabras con direcciones de memoria comprendidas entre 00000 y 3FFFF (caracterizadas por tener $A_{19} = 0$ y $A_{18} = 0$), únicamente la placa 0 lea o escriba sus contenidos desde o sobre el bus de datos.

Se necesita incluir en la placa 0 un pequeño circuito lógico de selección de placa, tal como se muestra en la Figura 2.30. De forma análoga se diseñarían los circuitos de selección de las restantes placas.

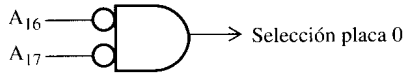


Figura 2.30: Circuito de selección de la placa 0

Al incluir los circuitos de selección en cada placa, éstas ya no serán idénticas. Por este motivo, para la selección de placas se suele utilizar un circuito de comparación tal como se muestra en la Figura 2.31. Este circuito se repite en todas las placas.

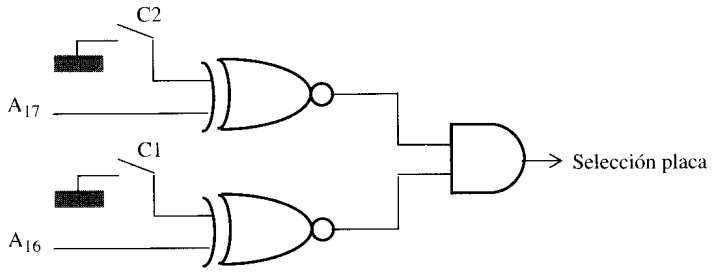


Figura 2.31: Circuito de comparación para la selección de placas

La asignación del espacio de direcciones a cada placa se consigue personalizando en cada una de ellas el estado de los conmutadores C_1 y C_2. Por ejemplo, si C_1 C_2 = 0 0, la línea de "selección de placa" vale 1 cuando $A_{19}A_{18}$ = 0 0 y por consiguiente se asignan las direcciones 00000 - 3FFFF (es decir los primeros 256*K*) a la placa 0. La selección de las restantes placas se realiza de forma similar. Sin embargo esta selección se puede incluir en la placa base. De esta forma todas las placas son idénticas y pueden conectarse en cualquiera de las ranuras de conexión sin tener que realizar la personalización.

Sección 2-4

Memorias caché

En todos los ciclos de instrucción la CPU accede a memoria al menos una vez para buscar la instrucción y frecuentemente realiza múltiples accesos para leer los operandos o almacenar los resultados. La velocidad con que la CPU puede ejecutar instrucciones está claramente limitada por el tiempo de ciclo de memoria. Esta limitación plantea un problema importante, debido a que las velocidades de funcionamiento de la CPU y de la memoria principal son diferentes. Se conoce como *tiempo de ciclo del procesador* el tiempo que la CPU necesita para realizar una de sus operaciones básicas, tales como leer de memoria o escribir en memoria, ejecutar el código de operación de una instrucción, comprobar si existen interrupciones, etc.

Surge nuevamente la necesidad de establecer un compromiso entre velocidad, coste y capacidad. Idealmente, se debería diseñar la memoria principal utilizando la misma tecnología que la empleada en los registros internos de la CPU, con lo que se conseguirían tiempos de ciclo de memoria y del procesador comparables. Esta estrategia, aunque conceptualmente simple y directa, es demasiado costosa. La solución estriba en colocar una memoria pequeña y rápida entre la CPU y la memoria principal (M_p): esta memoria se conoce como *memoria caché* (M_{ca}).

Este concepto se muestra en la Figura 2.32. Hay una memoria principal relativamente grande y lenta, conjuntamente con una memoria caché pequeña y rápida. Con esta estructura se utiliza, de forma inteligente, el "principio de localidad" ya mencionado al hablar de jerarquía de memorias.

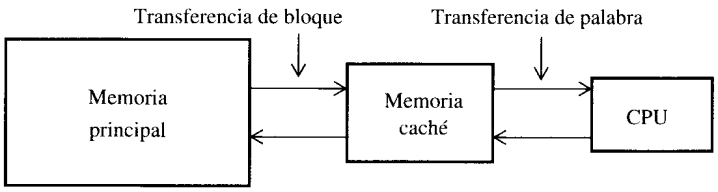


Figura 2.32: Memoria caché y memoria principal

La memoria caché almacena una copia de ciertas partes de la memoria principal. Cuando la CPU intenta leer una palabra de memoria en primer lugar comprueba si la palabra deseada está ya en la memoria caché. Si está, se lee la palabra desde la memoria caché. Si no está, se transfiere a la memoria caché un bloque de la memoria principal, que contiene un determinado número de palabras. Debido al fenómeno de la localidad de las referencias a memoria, es probable que cuando se transfiere un bloque de datos a la memoria caché, las futuras llamadas a memoria se hagan a otras palabras contenidas en el bloque transferido. Para el estudio de las memorias cachés es conveniente precisar los conceptos de bloque, acierto y fallo (ver Figura 2.33):

1) *Bloque*. Es la cantidad mínima de información que puede estar presente o no en M_p y M_{ca}.
2) *Acierto*. Cuando la CPU necesita una palabra de la M_p y la encuentra en la M_{ca}, se dice que se produce un acierto
3) *Fallo*. Si la palabra solicitada por la CPU a la M_p no está en la M_{ca} se contabiliza como un fallo.

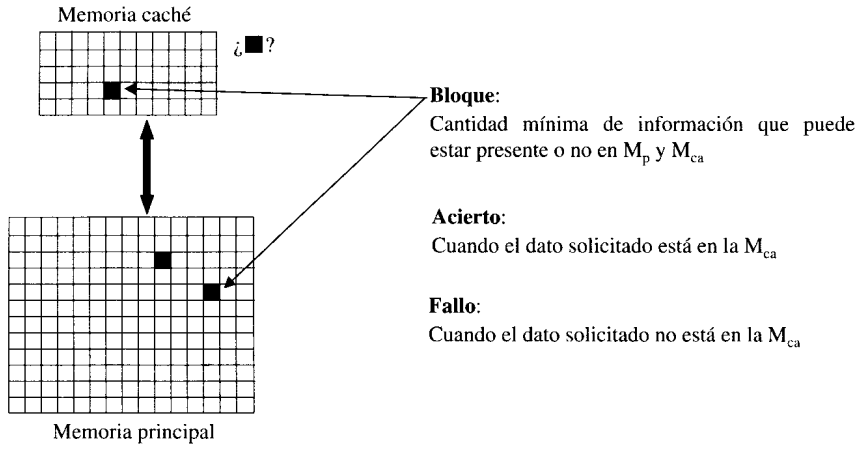


Figura 2.33: Conceptos de bloque, acierto y fallo de una memoria caché

En la Figura 2.34 se muestra la estructura de un sistema M_p-M_{ca}. La memoria principal consta de $N = 2^n$ palabras y cada palabra se puede referenciar mediante una dirección única de n bits.

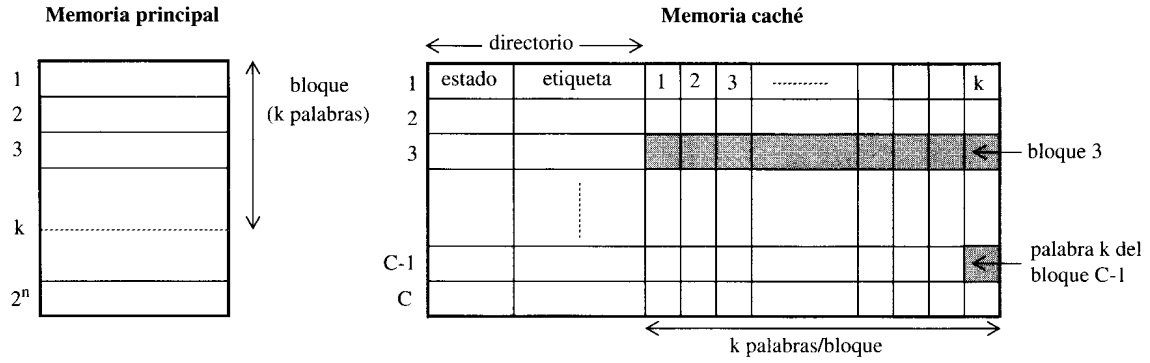


Figura 2.34: Estructura memoria principal-memoria caché

Con el fin de efectuar la transformación entre la memoria principal y la memoria caché, se considera que la memoria principal está constituida por una serie de bloques de longitud fija, de k palabras/bloque. Esto es, hay $M = 2^n/k$ bloques en la M_p. La memoria caché contiene C bloques de k palabras cada uno y el número de bloques de la memoria caché es considerablemente menor que el número de bloques que hay en la memoria

principal ($C << M$). Esto implica que la memoria caché tiene que disponer de un *directorio* que consta de *etiquetas* que permiten identificar qué bloque de la M_p se encuentra en cada momento en la M_{ca}. Los bits de *estado* sirven para indicar, entre otras cosas, si la información que contiene un bloque de caché es válida o no. Una vez vista la estructura de un sistema con memoria caché se puede analizar como se realiza en general la operación de lectura en una M_{ca} (ver Figura 2.35).

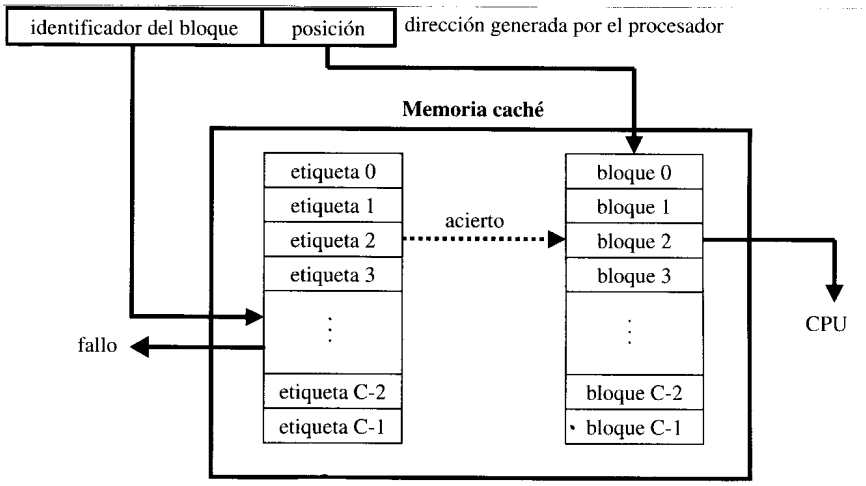


Figura 2.35: Acceso a la mememoria caché en una operación de lectura

La CPU genera una dirección para llevar a cabo un acceso a una determinada palabra de la M_p. Esta dirección desde el punto de vista de la M_{ca} consta de las dos partes. Una parte es el campo *identificador de bloque* que especifica el bloque donde se encuentra la palabra pedida y la otra indica el campo de *posición* de la palabra dentro del bloque. Por ejemplo si los bloques contienen 8 palabras, el campo de posición tendrá 3 bits que son los necesarios para seleccionar una palabra entre las 8 que tiene un bloque. Al realizar la búsqueda pueden ocurrir dos situaciones diferentes:

1) Que el identificador del bloque buscado coincida con alguna de las etiquetas de los bloques almacenados en el directorio de la M_{ca}. En ese caso se dice que se ha producido un acierto y con los bits correspondientes al campo de posición se selecciona la palabra pedida dentro del bloque para entregársela a la CPU.

2) Que el identificador del bloque buscado no coincida con alguna de las etiquetas de los bloques almacenados en el directorio de la M_{ca}. Esto quiere decir que la palabra pedida no se encuentra en ese momento en la M_{ca} y en consecuencia se ha producido un fallo de acceso. En este caso para realizar la operación de lectura solicitada por el procesador es necesario acceder a la M_p. Además debido al principio de localidad antes de finalizar la operación de lectura se deberá actualizar la M_{ca} para que la próxima vez que se haga referencia a esa dirección o a otra próxima a ella se encuentren ya en la M_{ca}. Generalmente la actualización lleva consigo la sustitución de uno de los bloques de la M_{ca} así como la modificación de la etiqueta correspondiente.

La Figura 2.36 muestra el diagrama de flujo de la operación de lectura en M_{ca} que se acaba de describir.

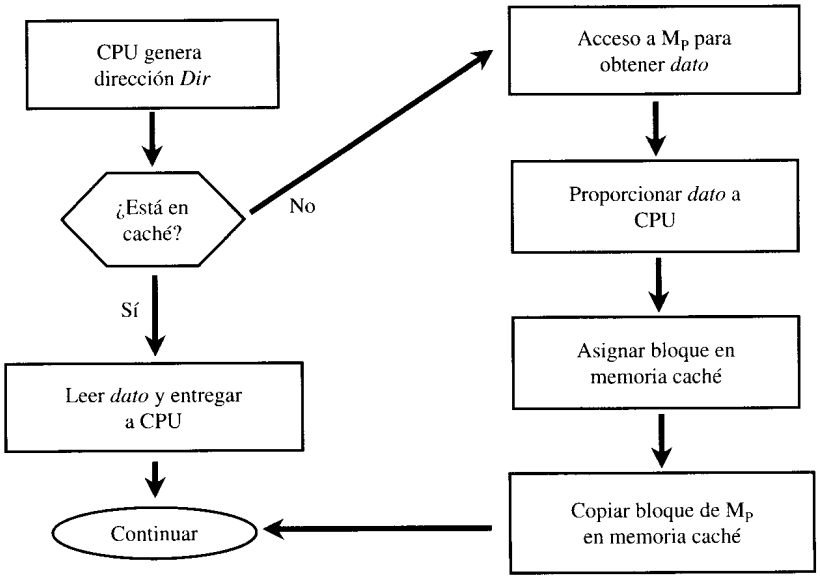


Figura 2.36: Diagrama de flujo de la operación de lectura de una memoria caché

Aunque hay gran diversidad de memorias caché, existen unos ciertos criterios básicos de diseño que sirven para clasificar y diferenciar las distintas arquitecturas de este tipo de memorias. La Tabla 2.9 da una relación de estos criterios.

Capacidad	1*K*, 4*K*, 16*K*, 32*K*, 64*K*,128*K*, 256*K*, ...
Organización de la memoria caché	Directa Totalmente asociativa Asociativa por conjuntos
Mecanismos de búsqueda	Por demanda Con anticipación Selectiva
Algoritmo de reemplazamiento	Utilizado menos recientemente (LRU) Más antigua (FIFO) Utilizado menos frecuentemente (LFU) Aleatorio
Estrategia de escritura	Escritura inmediata Post-escritura Escritura única
Tamaño del bloque	4, 8, 16, 32, ... palabras
Número de cachés	cachés de un nivel, caches de dos niveles, ...

Tabla 2.9: Elementos de diseño de una memoria caché

2.4.1 Rendimiento de una memoria caché

El rendimiento de una memoria caché se mide frecuentemente en términos de una cantidad llamada *tasa de acierto* (*h*).

$$h = \frac{\text{Número de veces que la palabra solicitada se encuentra en la } M_{ca}}{\text{Número total de referencias}}$$

Es decir *h*, que matemáticamente es una probabilidad, representa la razón entre el número de aciertos y el número total de referencias a memoria de la CPU (aciertos + fallos). Esta relación se suele medir experimentalmente ejecutando una serie de programas representativos en el computador y calculando el número de aciertos y de fallos durante un intervalo de tiempo dado.

En sistemas bien diseñados se suelen conseguir tasas de aciertos de 0,9 e incluso superiores. Este valor tan elevado verifica la validez del principio de localidad. La *tasa de fallos* es lógicamente 1-*h*. El tiempo de acceso medio t_a al sistema conjunto de memoria ($M_p + M_{ca}$) viene dado por la siguiente expresión:

$$t_a = ht_{ca} + (1 - h)t_p \tag{2.1}$$

t_{ca} = tiempo de acceso medio a la M_{ca} $\quad$ t_p = tiempo de acceso medio a la M_p

para la determinación de t_a en (2.1) se ha supuesto que el primer acceso debe hacerse a la memoria caché antes de realizar un acceso a la memoria principal. Los accesos hechos a M_p incrementan el tiempo de acceso en $(1 - h)t_p$.

En un sistema con memoria caché, la razón $\tau = \frac{t_{ca}}{t_p}$ suele estar entre 0,1 y 0,5. Este parámetro τ define la relación que existe entre el tiempo de acceso medio a la M_{ca} y el tiempo de acceso medio a la M_p. Valores típicos para t_p y t_{ca} pueden ser 100 ns y 20 ns respectivamente, lo que da un valor para τ de 0,2. Se puede definir ahora un cierto *índice de mejora* λ que se consigue al introducir en la jerarquía de memorias una memoria caché.

El parámetro λ define la relación que existe entre t_p (tiempo de acceso medio a la M_p) y t_a (tiempo de acceso medio al sistema conjunto de $M_p + M_{ca}$). Operando en la ecuación (2.1) y teniendo en cuenta la definición de τ se obtiene:

$$\lambda = \frac{t_p}{t_a} = \frac{1}{1 - h(1 - \tau)} \tag{2.2}$$

La Figura 2.37 muestra una gráfica de λ en función de la tasa de acierto *h*. Como se podía esperar, cuando $h = 0$ (todos los accesos se hacen a la M_p) el valor de λ es 1 y cuando $h = 1$ (todos los accesos se hacen a la M_{ca}) el valor de λ es $\frac{1}{\tau}$. La conclusión más importante que arroja la Figura 2.37 es que el índice de mejora es una función que es muy sensible al valor de la tasa de acierto *h*.

Solamente cuando *h* se aproxima a una tasa de acierto del 90 % es cuando el efecto de la memoria caché se hace realmente significativo. Este resultado es consistente con lo que dicta el sentido común. Si *h* cae por debajo del 90 %, los accesos a la M_p consumen una cantidad desproporcionada de tiempo y los accesos rápidos a la M_{ca} tienen poco efecto sobre el rendimiento del sistema.

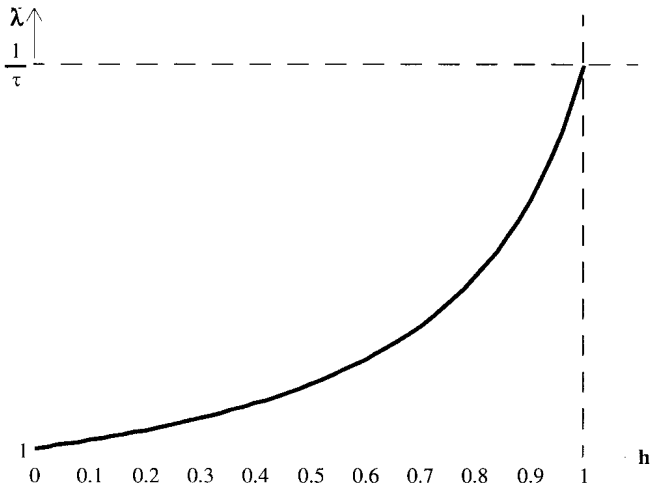


Figura 2.37: Índice de mejora λ de la M_{ca} en función de la tasa de acierto h

2.4.2 Capacidad de la memoria caché

El tamaño de la memoria caché (número de palabras) plantea un cierto compromiso. Por un lado, debería ser lo suficientemente pequeña como para que el coste medio por bit de información almacenado en la memoria interna del computador (memoria caché + memoria principal) estuviese próximo al de la memoria principal. Por otro, tendría que ser lo bastante grande como para que el tiempo de acceso medio total fuese lo más próximo posible al de la memoria caché. No obstante, hay algunas otras motivaciones para tratar de minimizar el tamaño de una memoria caché. Cuanto más grande sea la memoria caché más compleja será la lógica asociada con su direccionamiento. El resultado neto es que las memorias caché de gran capacidad tienden a ser más lentas que las más pequeñas, incluso empleando en ambas la misma tecnología de fabricación de circuitos integrados. También su capacidad tiende a estar limitada por el espacio físico que se le asigna, en muchos casos dentro del mismo circuito integrado que el procesador.

Una serie de estudios empíricos sugieren que el tamaño óptimo de una memoria caché está situado entre $1K$ y $512K$. Debido a que el rendimiento de la memoria caché es muy sensible al tipo de trabajo desarrollado, resulta imposible dar un valor exacto para el tamaño óptimo de una memoria caché.

2.4.3 Organización de la memoria caché

Existen diversas formas de como se puede organizar internamente una caché para almacenar su información. En todos los casos, la CPU referencia a la caché con la dirección de memoria principal del dato que necesita. Por este motivo cualquier organización de memoria caché debe utilizar esta dirección para encontrar el dato si lo tiene almacenado o indicarle al procesador cuando ha ocurrido un fallo. El problema de hacer corresponder la información mantenida en la memoria principal en su equivalente en la caché debe de implementarse totalmente en hardware si se desea conseguir un rendimiento óptimo en la operación del sistema. Básicamente la organización de la memoria caché consiste en establecer la *función de correspondencia, que asigna a los bloques de la memoria principal posiciones definidas en la memoria caché.* Para el cálculo de dicha función de correspondencia se emplean tres técnicas básicas:

- Directa
- Totalmente asociativa
- Asociativa por conjuntos

A continuación se examinan cada una de ellas utilizando el mismo ejemplo en las tres. El ejemplo incluye los siguientes elementos:

a) La memoria caché puede almacenar $512 = 2^9$ bytes.

b) Los datos se transfieren entre la memoria principal y la memoria caché en bloques de $k = 8$ bytes. Esto quiere decir que la memoria caché está organizada en $C = 64$ bloques de $k = 8$ bytes cada uno.

c) La memoria principal consta de $N = 32K$ bytes y cada una de ellos requiere $n = 15$ bits para especificar su dirección ($2^{15} = 32K$).

A efectos de la transformación se puede considerar que la memoria principal consta de $M = 4K$ bloques con $k = 8$ bytes/bloque. Como existen menos bloques en la memoria caché que bloques en la memoria principal, se necesita un algoritmo para calcular que bloque de la memoria principal ocupa un bloque dado de la memoria caché.

Correspondencia directa

Es la técnica más simple de todas. Cada bloque de la memoria principal se transforma en un único bloque de la memoria caché, tal como se muestra en la Figura 2.38.

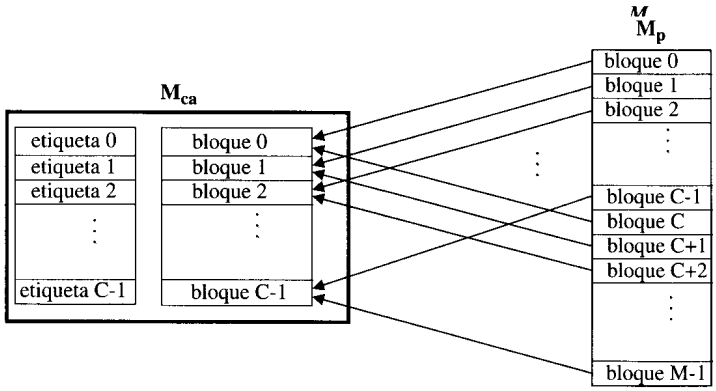


Figura 2.38: Asignación de bloques de la M_p en la M_{ca} con correspondencia directa

La función de transformación es:

$$i = j \text{ módulo } C \tag{2.3}$$

donde:

i = Número de bloque asignado en la M_{ca} al bloque de la M_p

j = Número de bloque de la M_p

C = Número de bloques que tiene la M_{ca}

En la Tabla 2.10 se muestra de acuerdo con la ecuación (2.3) la correspondencia entre los bloques de la memoria caché y de la memoria principal.

Bloque de M_{ca}	Bloque de M_p asignado
0	0, C, 2C, ..., M - C
1	1, C + 1, 2C + 1, ..., M - C + 1
⋮	⋮
C - 1	C - 1, 2C - 1, 3C - 1, ..., M - 1

Tabla 2.10: Correspondencia entre los bloques de M_{ca} y M_p

La Figura 2.39 ilustra el mecanismo general de la correspondencia directa.

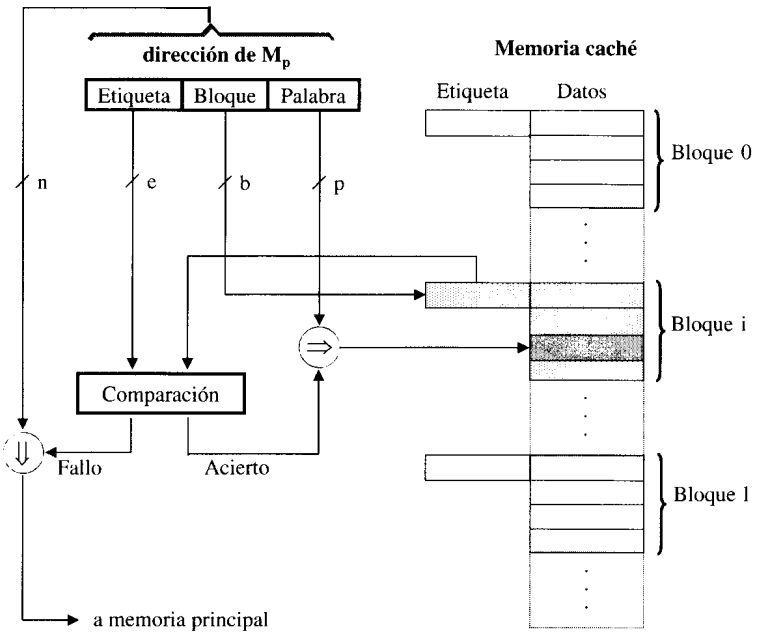


Figura 2.39: Organización de una memoria caché con correspondencia directa

Cuando la CPU genera una dirección para acceder a una palabra de la M_p, su formato desde el punto de vista de la M_{ca} se divide en tres campos: *etiqueta*, *bloque* y *palabra*. El número de bits de cada uno de estos campos (se supone, como es habitual que k y C son potencias de 2) viene dado por las relaciones siguientes:

campo palabra: $p = \log_2(k)$ *campo bloque*: $b = \log_2(C)$ *campo etiqueta*: $e = n - p - b$ (2.4)

El funcionamiento de la correspondencia directa se puede describir en los términos que siguen:

1) Con el campo bloque se identifica un bloque de la M_{ca}

2) Se compara la etiqueta de ese bloque de la M_{ca} con el campo etiqueta (de la dirección solicitada por el procesador). Si coinciden entonces con el campo palabra se selecciona la palabra pedida dentro del bloque y se le entrega a la CPU. Si al comparar las etiquetas no coinciden quiere decir que no se encuentra en la M_{ca} y por lo tanto se produce un fallo y habrá que ir a buscar la palabra a M_p.

En la Tabla 2.11 se resumen los parámetros del ejemplo propuesto y los de la organización de la memoria caché con correspondencia directa.

Tamaño de la M_p	N = 32 K	N° de bits para direccionar la M_p	n = 15
Tamaño de la M_{ca}	512 bytes	N° de bits del campo de palabra	p = 3
Palabras/bloque	k = 8	N° de bits del campo de bloque	b = 6
N° de bloques en la M_p	M = 4 K	N° de bits del campo de etiqueta	e = 6
N° de bloques en la M_{ca}	C = 64		

Tabla 2.11: Parámetros del ejemplo propuesto con organización con correspondencia directa

En el ejemplo propuesto (ver Figura 2.40), la función de transformación se realiza fácilmente utilizando los 15 bits de la dirección. Los 3 bits menos significativos (los que están más a la derecha) sirven para localizar una palabra dentro de un bloque de la M_p. Los restantes 12 bits especifican uno de los $2^{12} = 4K$ bloques de la memoria principal. La lógica que lleva incorporada la memoria caché interpreta estos 12 bits como una etiqueta de 6 bits (la parte más significativa) y los otros 6 bits (la parte menos significativa) identifican un bloque dentro de la memoria caché. Al mismo tiempo, estos 12 bits determinan el número del bloque de la memoria principal que se transfiere al correspondiente bloque de la memoria caché.

Así, los bloques 0000, 0100, ..., 7700 (los valores se representan en base octal) de la memoria principal se transforman en el bloque 0 de la memoria caché; los bloques 0001, 0101, ..., 7701 se transforman en el bloque 1 de la memoria caché, etc. La utilización de una parte de la dirección como campo de bloque proporciona una transformación única de cada bloque de la memoria principal en un bloque de la memoria caché. Cuando se asigna un bloque de M_p a un bloque de M_{ca}, es necesario ponerle una marca para distinguirlo de otros bloques que podrían transformarse en el mismo bloque. Los 6 bits más significativos de la dirección que constituyen el campo de etiqueta cumplen esta misión, de manera que los bloques 0000, 0100, ..., 7700 tienen como número de etiqueta 00, 01, ..., 77 respectivamente.

Una operación de lectura se realiza de la forma siguiente (ver Figura 2.40). A la memoria caché se le presenta una dirección de 15 bits. Los 6 bits del campo bloque se utilizan como índice dentro de la memoria caché para acceder a un bloque en particular. Si los 6 bits del campo etiqueta de la dirección coinciden con la etiqueta de ese bloque de la M_{ca}, se usan los 3 bits del campo palabra para seleccionar una de las 8 palabras que contiene. Si no coinciden se emplean, los 12 bits de los campos etiqueta y bloque conjuntamente, para localizar un bloque en la memoria principal. La dirección correspondiente está constituida por los 12 bits anteriores concatenados con 3 bits puestos a cero en la parte menos significativa. De esta manera se accede a 8 palabras que comienzan en una frontera de bloque.

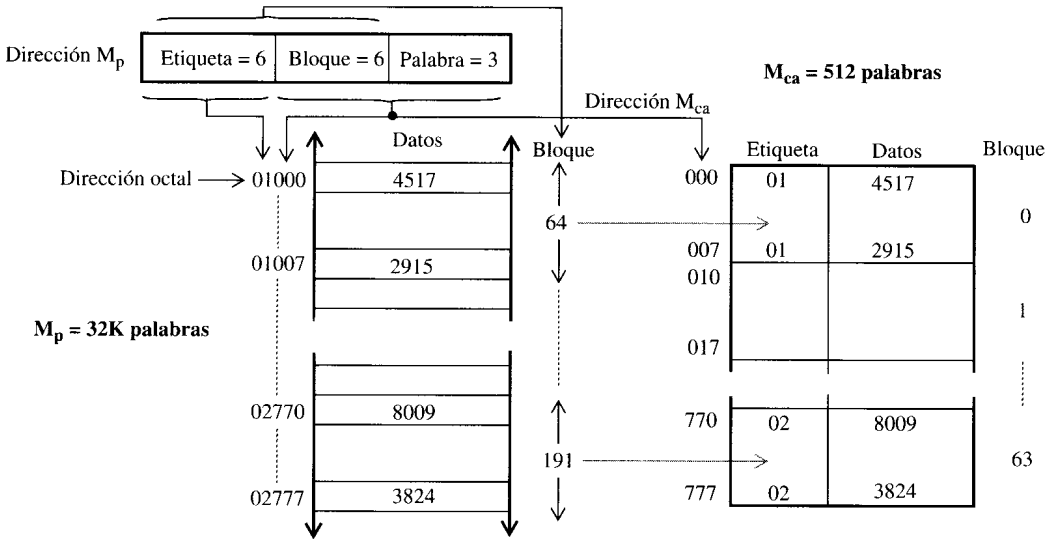


Figura 2.40: Ejemplo de correspondencia directa

La técnica de correspondencia directa es simple y poco costosa de realizar. Su principal desventaja es que cualquier bloque dado tiene asignada una posición fija en la memoria caché. Así, si ocurre que un programa efectúa repetidas referencias a palabras de dos bloques diferentes que se transforman en el mismo bloque de la M_{ca}, estos bloques se estarán moviendo continuamente entre la memoria caché y la memoria principal, con la consiguiente pérdida de rendimiento del sistema.

Correspondencia totalmente asociativa

Esta técnica subsana el inconveniente que se acaba de mencionar al estudiar la correspondencia directa, ya que permite que un bloque de memoria principal se cargue en cualquier bloque de la memoria caché, tal como se muestra en la Figura 2.41.

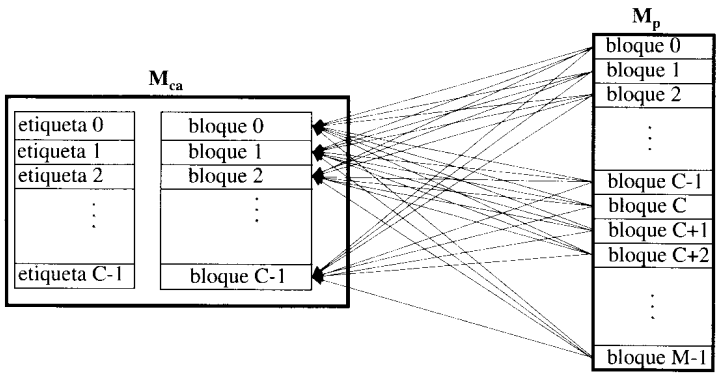


Figura 2.41: Asignación de bloques de la M_p en la M_{ca} con correspondencia totalmente asociativa

La Figura 2.42 ilustra el mecanismo general de la correspondencia totalmente asociativa. Cuando la CPU genera una dirección para acceder a una palabra de la M_p, su formato desde el punto de vista de la M_{ca} se divide en dos campos: *etiqueta*, y *palabra*. El número de bits de cada uno de estos campos viene dado por las relaciones siguientes:

$$\textit{campo palabra}: p = \log_2(k) \qquad \textit{campo etiqueta}: e = n - p \tag{2.5}$$

El funcionamiento de la correspondencia totalmente asociativa se puede describir en los términos que siguen:

1) Para determinar si un bloque está ya en la M_{ca} es preciso que esta incorpore la lógica necesaria que le permita examinar de forma simultánea todas las etiquetas de su directorio y comprobar si el campo etiqueta de la dirección solicitada por el procesador coincide con alguna de ellas. Un bloque de memoria principal se puede almacenar en cualquier bloque de la memoria caché, a condición de que se almacene con él los *e* bits de su campo de etiqueta.

2) Si coinciden, con el campo palabra se selecciona la palabra pedida dentro del bloque y se le entrega a la CPU. Si al comparar las etiquetas no coinciden quiere decir que no se encuentra en la M_{ca} y por lo tanto se produce un fallo y habrá que ir a buscar la palabra a M_p.

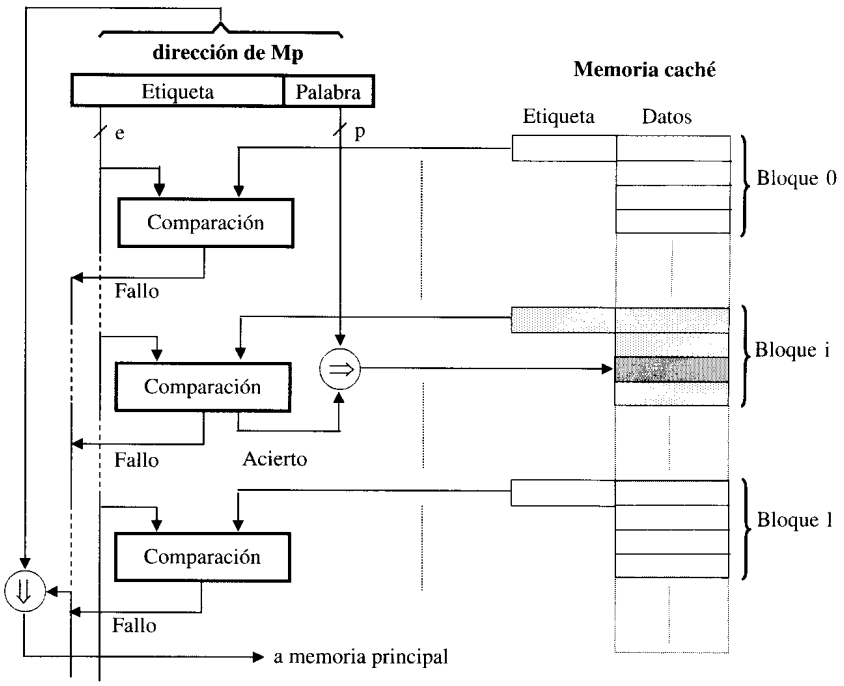


Figura 2.42: Organización de una memoria caché con correspondencia totalmente asociativa

En la Tabla 2.12 se resumen los parámetros del ejemplo propuesto y los de la organización de la memoria caché con correspondencia totalmente asociativa y en la Figura 2.43 se muestra el ejemplo que se viene desarrollando a lo largo de esta sección.

Tamaño de la M_p	N = 32 K	N° de bloques en la M_{ca}	C = 64
Tamaño de la M_{ca}	512 bytes	N° de bits para direccionar la M_p	n = 15
Palabras/bloque	k = 8	N° de bits del campo de palabra	p = 3
N° de bloques en la M_p	M = 4 K	N° de bits del campo de etiqueta	e = 12

Tabla 2.12: Parámetros del ejemplo propuesto con organización con correspondencia totalmente asociativa

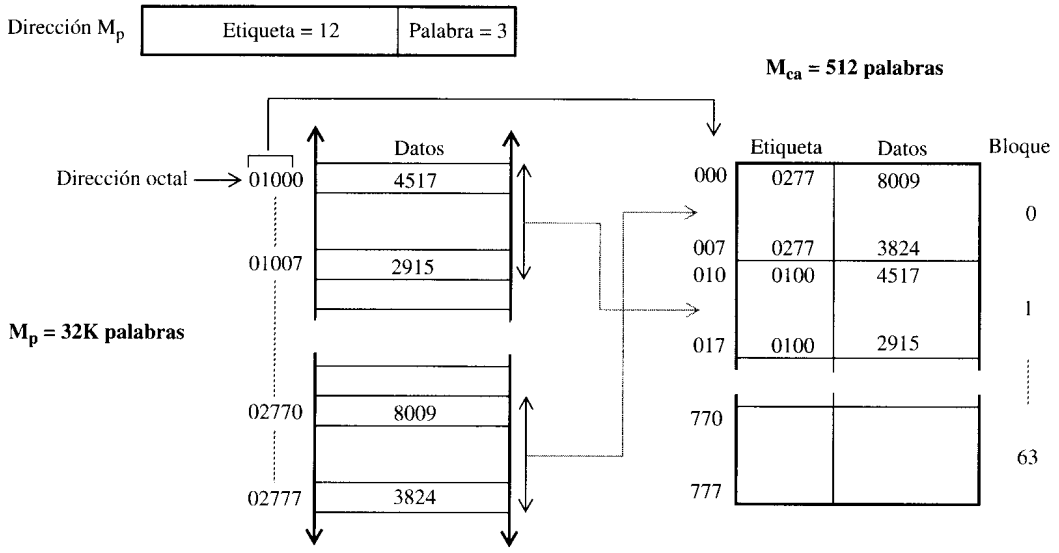


Figura 2.43: Ejemplo de correspondencia totalmente asociativa

Cuando se lee un nuevo bloque con la correspondencia totalmente asociativa es posible decidir por cuál se va a sustituir en la memoria caché. Los algoritmos de reemplazamiento, que se ven posteriormente, se diseñan con el fin de optimizar la probabilidad de encontrar la palabra solicitada en la M_{ca}. La principal desventaja de este procedimiento es la necesidad de una circuitería, bastante compleja, para examinar en paralelo los campos de etiqueta de todas los bloques de la memoria caché.

Correspondencia asociativa por conjuntos

Esta técnica es un compromiso que trata de aunar las ventajas de los dos métodos anteriores. Para su aplicación, la memoria caché se divide en q conjuntos, cada uno de los cuales consta de r bloques. Se tiene que:

$$C = q \times r$$

$$i = j \text{ módulo } q$$

donde

q = número de conjuntos en los que se divide la M_{ca}

r = número de bloques que contiene cada uno de los conjuntos

i = número de bloque de la M_{ca}

j = número de bloque de la M_p

C = número de bloques que contiene la M_{ca}

Cada bloque de la memoria principal puede ubicarse en uno de los *r* bloques de la memoria caché que componen cada conjunto. Este número *r* define el *grado de asociatividad* de la M_{ca}. Con este algoritmo, el bloque *j* de la M_p se puede almacenar en un bloque cualquiera del conjunto que tiene asociado en la M_{ca}. En la Figura 2.44 se muestra un ejemplo para el caso en que $r = 2$ (por lo que $q = C/2$). La correspondencia asociativa por conjuntos se puede ver como una correspondencia directa entre los bloque de la M_p y los conjuntos de la M_{ca} y como una correspondencia totalmente asociativa entre los bloques de un mismo conjunto.

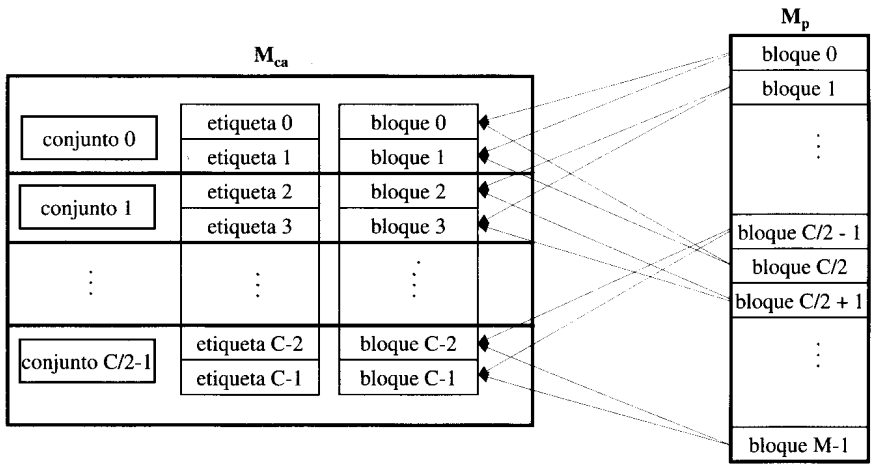


Figura 2.44: Asignación de bloques de la M_p en la M_{ca} con correspondencia asociativa por conjuntos

La Figura 2.45 ilustra el mecanismo general de la correspondencia asociativa por conjuntos. Cuando la CPU genera una dirección para acceder a una palabra de la M_p, su formato desde el punto de vista de la M_{ca} se divide en tres campos: *etiqueta*, *conjunto* y *palabra*. El número de bits de cada uno de estos campos viene dado por las relaciones siguientes:

campo palabra: $p = \log_2(k)$ *campo conjunto*: $c = \log_2(q)$ *campo etiqueta*: $e = n - p - c$ (2.6)

Los *c* bits del campo conjunto especifican uno de entre $q = 2^c$ conjuntos en los que se divide la M_{ca}. Los $e + c$ bits de los campos de etiqueta y conjunto especifican uno de los $M = 2^{n-p}$ bloques de M_p. Cada bloque en cada conjunto tiene una etiqueta almacenada que conjuntamente con el número del conjunto completa la identificación del bloque de la M_{ca}. El funcionamiento de la correspondencia asociativa por conjuntos se

puede describir en los términos que siguen:

1) En primer lugar se utiliza el número de conjunto de la dirección solicitada por el procesador para acceder al conjunto correspondiente de la M_{ca}.

2) Se comparan simultáneamente todas las etiquetas del conjunto seleccionado con la etiqueta de la dirección solicitada. Si coinciden entonces se accede a la palabra pedida en la M_{ca}, en caso contrario se produce un fallo y habrá que ir a buscar la palabra a M_p.

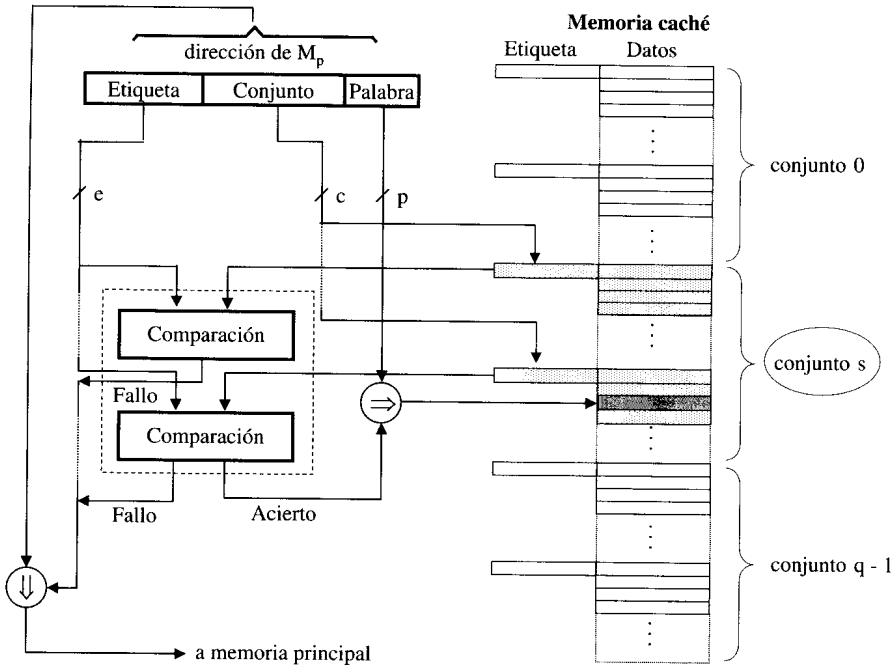


Figura 2.45: Organización de una memoria caché con correspondencia asociativa por conjuntos

En el ejemplo que se considera (ver Figura 2.46) hay 32 conjuntos con 2 bloques/conjunto ($q = 32 = 2^5$, $r = 2$). Los cinco bits del campo conjunto ($q = 2^5 \Rightarrow c = 5$), identifican a un único conjunto de dos bloques dentro de la M_{ca}. También indica el número del bloque (módulo 2^5) de la M_p que se transfiere al bloque correspondiente de la memoria caché. Así los bloques 0000_8, 0040_8, 0100_8, 0140_8 ..... (expresados en base octal) de la memoria principal se transforman en el conjunto 0 de la memoria caché ($i = 0$), ya que:

$$0000_8 = 0_{10} \Rightarrow i = 0_{10} \bmod 32_{10} = 0$$

$$0040_8 = 32_{10} \Rightarrow i = 32_{10} \bmod 32_{10} = 0$$

$$0100_8 = 64_{10} \Rightarrow i = 64_{10} \bmod 32_{10} = 0, \text{ etc.}$$

Como este conjunto contiene 2 bloques, los bloques de la M_p se pueden cargar en cualquiera de ellos. Ninguno de los dos bloques que se transforman en el mismo conjunto de la M_{ca} pueden tener números de etiquetas iguales. En una operación de lectura, los cinco bits del campo conjunto se utilizan para determinar

que conjunto de dos bloques hay que examinar. Después se comprueban los dos bloques del conjunto que corresponda para ver si hay coincidencia con el número de etiqueta de la dirección a la que se desea acceder.

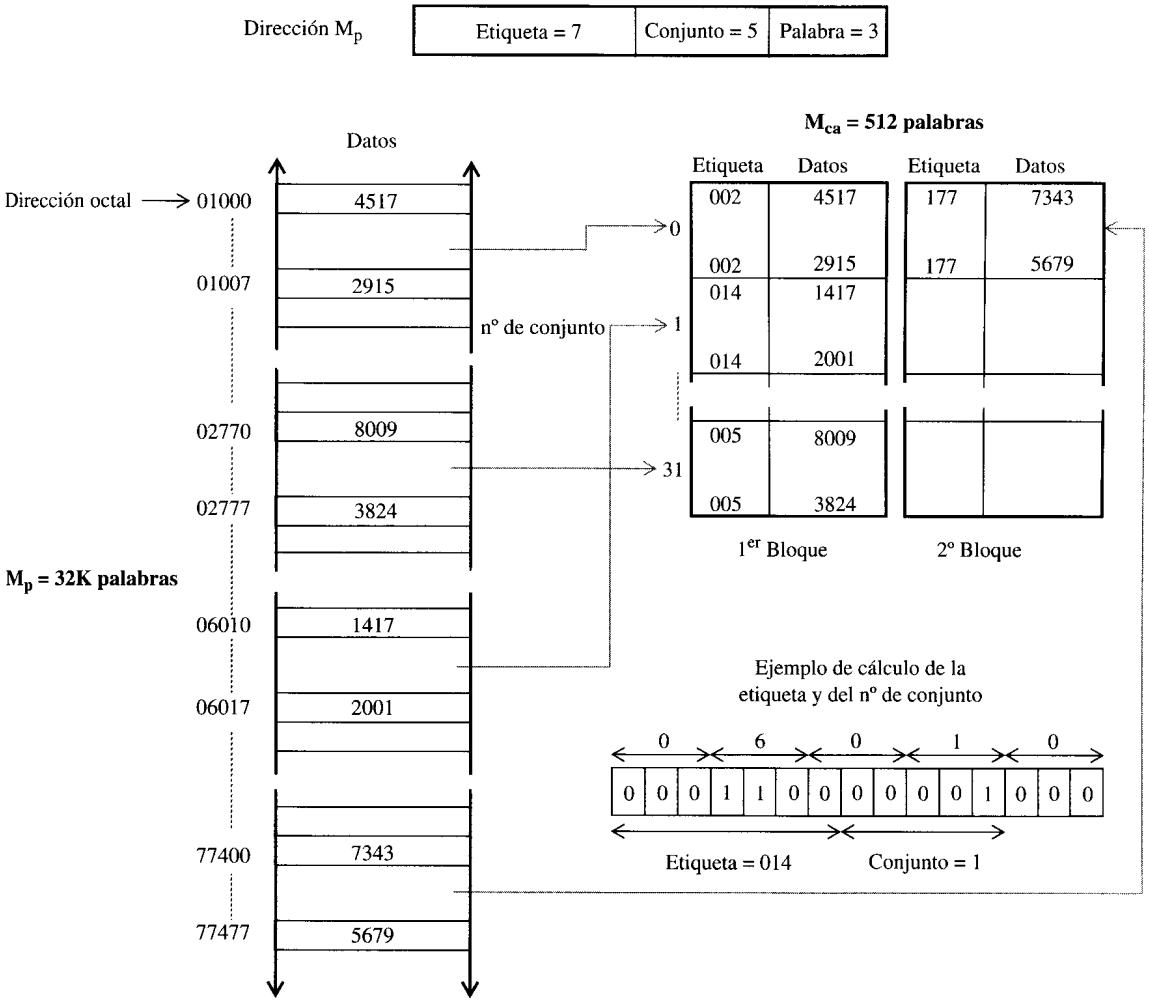


Figura 2.46: Correspondencia asociativa por conjuntos

La correspondencia asociativa por conjuntos contiene las dos técnicas anteriores como casos particulares:

a) $q = C$, $r = 1 \Rightarrow$ correspondencia directa

b) $q = 1$, $r = C \Rightarrow$ correspondencia totalmente asociativa

La utilización de $q = C/2$, $r = 2$ (es decir, 2 bloques por conjunto), es la organización más usual en la correspondencia asociativa por conjuntos. En este caso, se mejora significativamente el rendimiento de la caché frente a la transformación directa. También se emplea $q = C/4$, $r = 4$ (es decir, 4 bloques por conjunto)

que proporciona una relativa mejora a un coste moderado. A partir de este valor, el beneficio marginal que se obtiene al aumentar el número de particiones por conjunto no compensa, por el mayor coste que supone.

2.4.4 Algoritmos de reemplazamiento

Cuando un nuevo bloque se transfiere a la memoria caché debe sustituir a uno de los ya existentes. En el caso de la correspondencia directa la partición está determinada unívocamente y por lo tanto no se puede realizar ninguna elección. Por el contrario, en las técnicas de tipo asociativo es necesario un *algoritmo de reemplazamiento.* Este mecanismo de sustitución debe realizarse totalmente por hardware y preferiblemente de manera que la selección se haga toda ella durante el ciclo de memoria principal de búsqueda del nuevo bloque. Lo ideal sería que el bloque sustituido no se necesitase utilizar otra vez en el futuro, sin embargo estas circunstancias no pueden conocerse a priori y la decisión de qué bloque hay que reemplazar debe basarse sobre hechos conocidos en el momento de tomarla. De forma general los algoritmos de sustitución de bloques de la M_{ca} se pueden clasificar en dos grandes grupos: los que están basados en su utilización y aquellos que no toman en cuenta esta consideración. No obstante en los dos casos un factor crítico es a menudo la cantidad de hardware que se necesita para implementar el algoritmo.

Para una M_{ca} con organización totalmente asociativa cualquier algoritmo de reemplazamiento que se base en la utilización de los bloques debe considerar como posibles candidatos todos los bloques en el momento de tomar la decisión. Por el contrario si la organización de la M_{ca} es asociativa por conjuntos sólo deberá considerar los bloques de un determinado conjunto (aunque será preciso mantener un cierto registro del uso relativo de los bloques en cada conjunto). Del razonamiento expuesto se sigue sin embargo que cualquiera que sea el algoritmo empleado (basado en la utilización de los bloques o no), hay generalmente menos bloques que considerar en una M_{ca} asociativa por conjuntos que en una totalmente asociativa y por lo tanto se requerirá un hardware más simple. El caso límite es cuando hay dos bloques/conjunto ($r = 2$) que sólo requiere un bit adicional por conjunto para indicar que bloque se debe reemplazar. El grado de asociatividad de las cachés, como ya se ha dicho anteriormente es pequeño ($r = 2$ ó 4). Entre los diferentes algoritmos que se han propuesto merecen destacarse los cuatro siguientes:

a) *Bloque elegido de forma aleatoria*

b) *Bloque utilizado menos frecuentemente.* Algoritmo LFU (Least Frequently Used)

c) *Bloque más antiguo en memoria caché.* Algoritmo FIFO (First In First Out)

d) *Bloque utilizado menos recientemente.* Algoritmo LRU (Least Recently Used)

Bloque elegido de forma aleatoria

Este método toma de forma aleatoria un bloque de la M_{ca} y lo sustituye por el bloque de la M_p que contiene a la palabra a la que se acaba de acceder. Es muy fácil de implementar en hardware y es más rápido que la mayoría de los otros algoritmos. Su desventaja obvia es que el bloque que es más probable que se vaya a utilizar otra vez tiene la misma probabilidad de ser sustituido que cualquier otro bloque. Esta desventaja se ve mitigada cuando el tamaño de la M_{ca} aumenta.

Bloque utilizado menos frecuentemente

Este método supone que los bloques que no se referencian con frecuencia no se necesitan tanto. Cada bloque de la M_{ca} incorpora un contador que almacena el número de veces que dicho bloque se ha utilizado desde que

fue transferido a la memoria caché. El bloque que tenga el mínimo valor en su contador es el que se sustituye. La ventaja que presenta este algoritmo de reemplazamiento es que el bloque que se utiliza frecuentemente es más probable que permanezca en la M_{ca} y no así el que no se referencia a menudo. Una desventaja que es consustancial a este método es que penaliza en exceso a los bloques que se han transferido recientemente a la M_{ca} ya que necesariamente tendrán un contador con un valor muy bajo a pesar de que es probable que serán utilizados otra vez en el futuro inmediato. Otra desventaja es que el método es más difícil de implementar en términos de hardware y es por lo tanto más costoso.

Bloque más antiguo en memoria caché

Este método elimina de la M_{ca} el bloque que ha permanecido durante más tiempo. El algoritmo de *primero en entrar primero en salir* se realiza de forma natural con una cola de direcciones de bloques, sin embargo se puede implementar más fácilmente con contadores. En una M_{ca} totalmente asociativa sólo es preciso un contador y para una M_{ca} asociativa por conjuntos se requiere un contandor por cada conjunto. En ambos casos los contadores deberán tener un número suficiente de bits para identificar completamente al bloque.

Bloque utilizado menos recientemente

Este método tiene el mejor rendimiento en relación con su coste cuando se le compara con las otras técnicas y es el que más se utiliza en los sistemas reales. La idea en la que se fundamenta el algoritmo es que un bloque que no se ha utilizado durante un largo período de tiempo tiene una probabilidad menor de ser utilizado en el futuro inmediato de acuerdo con el principio de localidad temporal. Así, este método retiene bloques en la M_{ca} que son más probables que puedan ser utilizados otra vez. Para lograrlo se emplea un mecanismo que almacena qué bloques han sido accedidos más recientemente. El bloque que se elimina de la M_{ca} es el que no se ha referenciado durante el período más largo de tiempo. Una forma de implementar esta técnica es asignar un contador a cada bloque de la M_{ca}. Cada vez que se accede a la M_{ca}, cada contador de bloque se incrementa en uno salvo el contador del bloque al que se accedió que se inicializa a cero. De esta forma el bloque cuyo contador tiene el máximo valor es el utilizado menos recientemente. El valor de cada contador indica así la *edad* de un bloque desde la última vez que se referenció.

El algoritmo para estos registros de envejecimiento se puede modificar para tomar en cuenta el hecho de que los contadores tienen un número fijo de bits y que sólo se precisa la edad relativa entre ellos. Si no se tuviese esto en cuenta se produciría un desbordamiento del contador que alteraría el resultado convirtiendo al bloque que más tiempo hace que se utiliza en el que menos. El algoritmo es el siguiente:

1) Cuando ocurre un acierto, el contador asociado con ese bloque se inicializa a cero, lo que indica que es el que se ha utilizado más recientemente. Todos los contadores que tienen un valor más pequeño que el que tenía el contador del bloque accedido se incrementan en uno. Todos los contadores que tienen un valor mayor que el que tenía el contador del bloque accedido no se modifican.

2) Cuando ocurre un fallo y la M_{ca} no está llena, el contador asociado con el bloque que proviene de la M_p se inicializa a cero y todos los otros contadores se incrementan en uno. Si la M_{ca} está llena, entonces se sustituye el bloque que tiene el contador con el máximo valor (p. ej. 3 para un contador de 2 bits) y se inicializa a cero su contador; todos los otros contadores se incrementan en uno.

Con este algoritmo el contador con el máximo valor identifica al bloque utilizado menos recientemente.

En la Tabla 2.13 se muestra un ejemplo de una M_{ca} asociativa por conjuntos con 4 bloques/conjunto ($q = 4$). Un contador de 2 bits es suficiente para cada bloque y C_0, C_1, C_2 y C_3, representan los contadores en un conjunto. Al comienzo del algoritmo todos los contadores se inicializan a cero. Se observa que cuando un bloque de la M_p sustituye a otro en la M_{ca} cambia el valor de la etiqueta que referencia de forma unívoca a cada bloque.

Nº acceso	Etiqueta	Acierto/Fallo	C_0	C_1	C_2	C_3	Acciones
			0	0	0	0	Inicialización
1	5	Fallo	0	1	1	1	Bloque 0 rellenado
2	6	Fallo	1	0	2	2	Bloque 1 rellenado
3	5	Acierto	0	1	2	2	Bloque 0 accedido
4	7	Fallo	1	2	0	3	Bloque 2 rellenado
5	8	Fallo	2	3	1	0	Bloque 3 rellenado
6	6	Acierto	3	0	2	1	Bloque 1 accedido
7	9	Fallo	0	1	3	2	Bloque 0 sustituido
8	10	Fallo	1	2	0	3	Bloque 2 sustituido

Tabla 2.13: Algoritmo LRU utilizando contadores

2.4.5 Estrategia de escritura

Para que pueda reemplazarse un bloque residente en la memoria caché, es necesario considerar si ha sido modificado en la M_{ca} pero no en la M_p. Si no lo ha sido, se puede reescribir sobre el bloque antiguo de la memoria caché. Si se ha modificado el bloque, significa que al menos se ha realizado una operación de escritura sobre una palabra de ese bloque de la memoria caché y se debe actualizar la memoria principal. Hay una serie de estrategias de escritura, cada una de ellas con un coste y un rendimiento asociado. En general se plantean dos problemas.

a) Más de un dispositivo puede tener acceso a la memoria principal. Por ejemplo, un módulo de E/S puede ser capaz de leer/escribir directamente a memoria. Si una palabra ha sido modificada solamente en la memoria caché, entonces la palabra de memoria principal correspondiente ya no es válida. Por otra parte, si el dispositivo de E/S lo que ha alterado es la memoria principal, no es válido el contenido de dicha palabra en la memoria caché.

b) Varias CPU's se conectan a un mismo bus y cada una de ellas tiene su propia memoria caché local. En este caso, si se cambia una palabra en una memoria caché podría invalidar una palabra en otras memorias cachés.

La técnica más simple se denomina *escritura inmediata* (write through). Consiste en efectuar todas las operaciones de escritura tanto en la memoria principal como en la memoria caché, lo que asegura que sus contenidos son siempre válidos. Cualquier otro módulo CPU-memoria caché puede supervisar el tráfico con la memoria principal, para mantener la consistencia dentro de su propia memoria caché. La principal

desventaja de esta técnica es que genera un tráfico elevado con la memoria y esto puede crear una saturación que disminuye significativamente el rendimiento del sistema.

Una técnica alternativa que minimiza las escrituras en memoria, consiste en realizar las actualizaciones sólo en la memoria caché. Este método se conoce como *post-escritura* (write back). Cuando se produce una modificación se pone a 1 un "bit de actualización" asociado con cada bloque de la memoria caché. Si se reemplaza un bloque se reescribe en la memoria principal si y sólo si el bit de actualización está a 1. El problema que hay con esta técnica es saber qué partes de la memoria principal no son válidas. Para obviar esta dificultad, los accesos de los módulos de E/S se permiten únicamente a través de la memoria caché.

En una organización de bus en la que más de un dispositivo (normalmente una CPU) tiene una memoria caché asociada y se comparte la memoria principal aparece un nuevo problema. Si se modifica una palabra de la memoria caché se invalida, además de la palabra correspondiente en la memoria principal, la misma palabra en otras memorias cachés (si ocurre que algunas de las memorias cachés contienen esa misma palabra). Incluso, con la estrategia de escritura inmediata, las otras memorias cachés pueden contener datos inválidos. Un sistema que evita este problema se dice que mantiene la coherencia de la M_{ca}.

2.4.6 Tamaño del bloque

Otro elemento en el diseño de una memoria caché es el tamaño del bloque. Cuando se transfiere un bloque de la memoria principal a la memoria caché, con la palabra deseada se mueve un conjunto de palabras adyacentes. Por esta razón, cuando se va aumentando el tamaño del bloque a partir de valores muy pequeños la tasa de acierto inicialmente aumentará a causa del principio de localidad, ya que se incrementa la probabilidad de que sean accedidos, en el futuro inmediato, datos próximos a la palabra referenciada. Sin embargo, a partir de un cierto tamaño del bloque la tasa de acierto comienza a disminuir, debido a que la probabilidad de utilizar la información contenida en el bloque se hace menor que la probabilidad de reusar la información que se tiene que reemplazar. Aparecen pues dos efectos específicos:

1) Cuanto mayor sea el tamaño de los bloques, menor será el número de éstos que es posible tener en la memoria caché. Como cada búsqueda de un bloque en la memoria principal lleva consigo su escritura en la memoria caché, si el número de bloques es pequeño aumenta la frecuencia de utilización del algoritmo de reemplazamiento de bloques.

2) Cuando crece el tamaño de un bloque, cada nueva palabra añadida a ese bloque estará a mayor distancia de la palabra requerida por la CPU, y por lo tanto es menos probable que sea necesitada a corto plazo.

La relación entre tamaño de bloque y tasa de acierto es compleja, y depende de las características de localidad de cada programa en particular. Por este motivo no se ha encontrado un valor óptimo definitivo, sin embargo, una elección razonable es un tamaño entre 4 y 8 palabras.

2.4.7 Número de cachés

Cuando la M_{ca} se integra en el propio procesador, puede resultar imposible aumentar su tamaño si el rendimiento del sistema no es el adecuado. En cualquier caso ampliar el tamaño de la M_{ca} puede producir una caché que sea más lenta que la original, Una alternativa que se está utilizando cada vez más es introducir una segunda caché de mayor capacidad que se localiza entre la primera caché y la memoria principal tal como se muestra en la Figura 2.47. Esta *caché de segundo nivel* se denomina algunas veces *caché secundaria*. Su

tamaño suele ser un orden de magnitud mayor que el de la caché de primer nivel, por ejemplo $512K$ y $64K$ para las M_{ca} de segundo y primer nivel respectivamente.

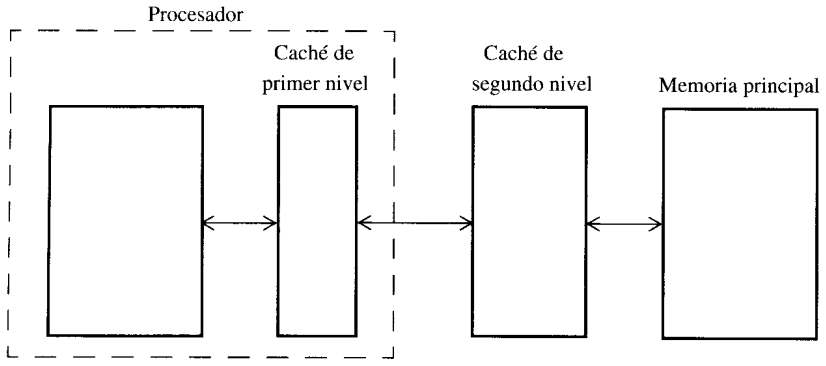


Figura 2.47: Memoria caché de dos niveles

En una referencia a memoria, el procesador accederá a la caché de primer nivel. Si la información no se encuentra allí (se dice que ocurre un fallo en la M_{ca} de primer nivel), se accede a la caché de segundo nivel. Si tampoco la información solicitada se encuentra allí (se dice que ocurre un fallo en la M_{ca} de segundo nivel) habrá que acceder a la M_p. El bloque de memoria correspondiente se transfiere a la caché de segundo nivel y luego a la del primer nivel de forma que existen al menos inicialmente dos copias del bloque de memoria. Esto quiere decir que todos los bloques transferidos a la caché de segundo nivel también existen en la caché de primer nivel. Esto se conoce como *principio de inclusión*. Por supuesto que las copias de los bloques de la caché de segundo nivel nunca se necesitarán cuando se encuentran en la caché del primer nivel.

La política de reemplazamiento y la estrategia de escritura practicada en ambas cachés normalmente suelen ser respectivamente el algoritmo LRU y la escritura inmediata para mantener las copias duplicadas. El tamaño del bloque de la caché de segundo nivel será igual o mayor que el tamaño del bloque de la cache de primer nivel ya que en el caso contrario si se produce un fallo en esta última se necesitarían transferir más de un bloque de la primera.

El rendimiento de una M_{ca} visto en el apartado 2.4.1 se puede extender a una caché de dos niveles. En este caso se obtiene la expresión siguiente:

$$t_a = [h_1 t_{ca1} + (1 - h_1) t_{ca2}] + (1 - h_2) t_p \qquad (2.7)$$

donde

t_{ca1} = tiempo de acceso a la M_{ca} de primer nivel

t_{ca2} = tiempo de acceso a la M_{ca} de segundo nivel

t_p = tiempo de acceso a la M_p

h_1 = tasa de acierto de la M_{ca} de primer nivel

h_2 = tasa de acierto combinada de las M_{ca} de primer y segundo nivel

para la determinación de t_a en (2.6) se ha considerado que las dos cachés son homogéneas. La tasa de acierto h_1 será mayor que la tasa de acierto h_2.

La mayoría de las familias de microprocesadores proporcionan cachés de segundo nivel que emplean controladores independientes para las cachés externas a la CPU. Así el Intel 486 dispone del circuito controlador de caché 8291 y el Pentium del 82491. Motorola también posee controladores de cachés análogos.

Sección 2-5

Memorias asociativas

Tal como se indicó en la sección 2-1 al comentar los diferentes métodos de acceso, una memoria asociativa se caracteriza por el hecho de que la identificación de la posición de memoria a la que se desea acceder se realiza especificando su contenido o parte de él. Por este motivo se denominan también *memorias direccionables por contenido* o *memorias CAM* (Content Addressable Memory).

La operación básica de recuperación de información en una memoria asociativa, consiste en seleccionar las palabras de memoria que contienen una cierta información denominada *clave* y extraer la información asociada a dicha clave en las palabras seleccionadas.

2.5.1 Ejemplo: Concepto de memoria asociativa

Sea una memoria asociativa de 4 palabras cuyo contenido es el que se muestra en la Figura 2.48.

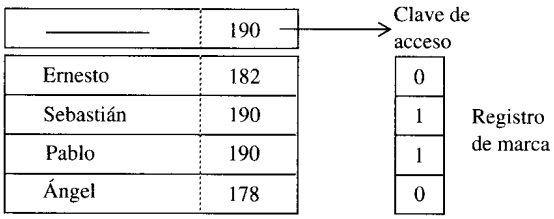


Figura 2.48: Ejemplo de memoria asociativa

La memoria contiene el nombre y la altura, expresada en cms, de un conjunto de personas. Si se desea conocer el nombre de las personas que miden 190 cm, se utilizará este valor como clave de acceso y el hardware de la memoria se encargará de actualizar un registro que se conoce como *registro de marca*. Este registro contiene un bit por cada palabra de la memoria y cada vez que se realiza una operación de búsqueda en la memoria mediante una clave de acceso determinada, se ponen a 1 los bits correspondientes a las palabras de memoria que contienen la clave especificada♦.

2.5.2 Estructura de una memoria asociativa

En la Figura 2.49 se muestra el diagrama de bloques de una memoria asociativa. Consiste en una matriz de celdas de memoria, con su lógica asociada, organizada en n palabras con m bits/palabra. El *registro argumento A* y el *registro de máscara K* tienen *m* bits cada uno y el *registro de marca M* consta de *n* bits.

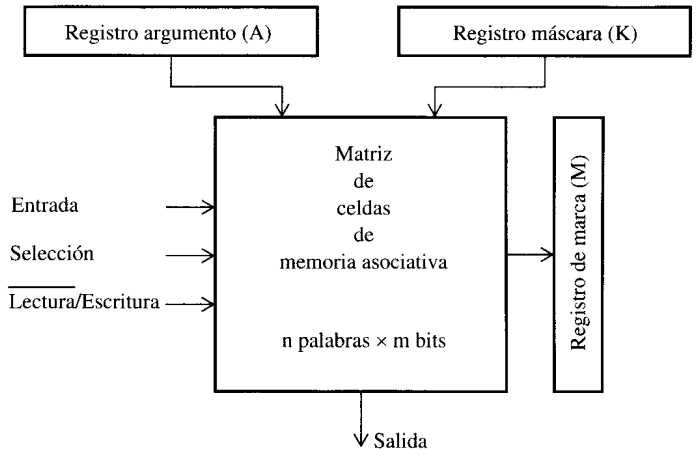


Figura 2.49: Diagrama de bloques de una memoria asociativa

Cada palabra de la memoria se compara en paralelo con el contenido del registro argumento, y se pone a 1 el bit del registro de marca asociado a aquellas palabras cuyo contenido coincide con el del registro argumento. Al final de este proceso, aquellos bits del registro de marca que están a 1 indican la coincidencia de las correspondientes palabras de la memoria asociativa y del registro argumento. El proceso de lectura se realiza mediante un acceso secuencial a las palabras de memoria cuyos bits en el registro de marca están a 1.

El registro de máscara proporciona la clave para seleccionar la parte que se desee en la palabra argumento. Si en el registro de máscara se ponen todos sus bits a 1, se comparan todas las palabras de la memoria asociativa con el registro argumento. En caso contrario, solamente se comparan aquellos bits del registro argumento cuya posición se corresponde con los bits que son 1 en el registro de máscara. En la Figura 2.50 se presenta un ejemplo.

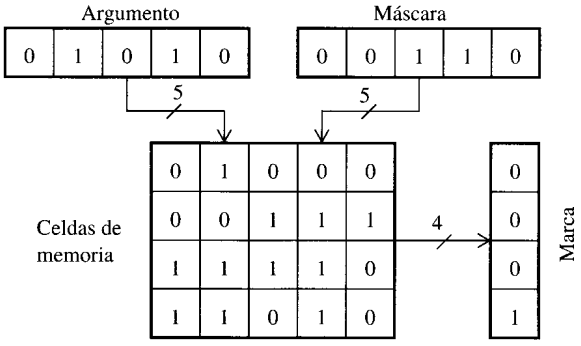


Figura 2.50: Ejemplo de utilización de los registros

La relación entre las celdas de almacenamiento y los registros externos en una memoria asociativa se muestra en la Figura 2.51. La celda C_{ij} corresponde al bit j de la palabra i.

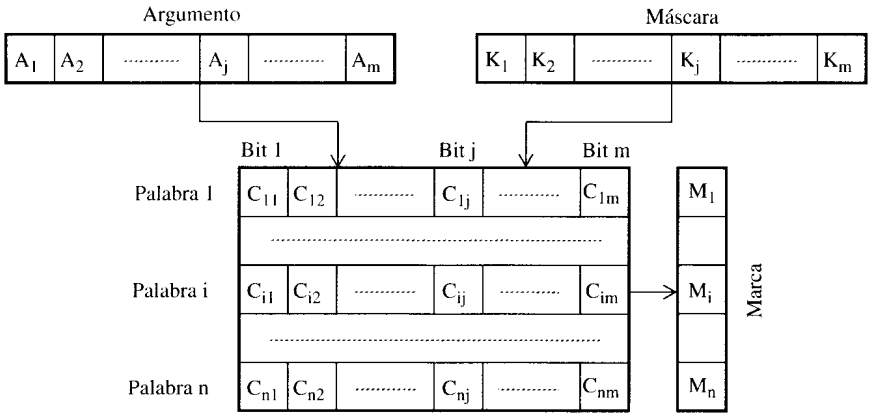


Figura 2.51: Memoria asociativa de n palabras × m bits

El bit A_j ($j = 1, 2, .., m$) del registro argumento se compara con todos los bits de la columna j si $K_j = 1$. Si existe coincidencia entre todos los bits sin máscara del registro argumento y los bits de la palabra i, se hace $M_i = 1$. En caso contrario $M_i = 0$. En la Figura 2.52 se da la estructura interna de una celda C_{ij}. Está formada por un elemento de memoria Q_{ij} y la circuitería lógica adecuada para hacer las funciones siguientes:

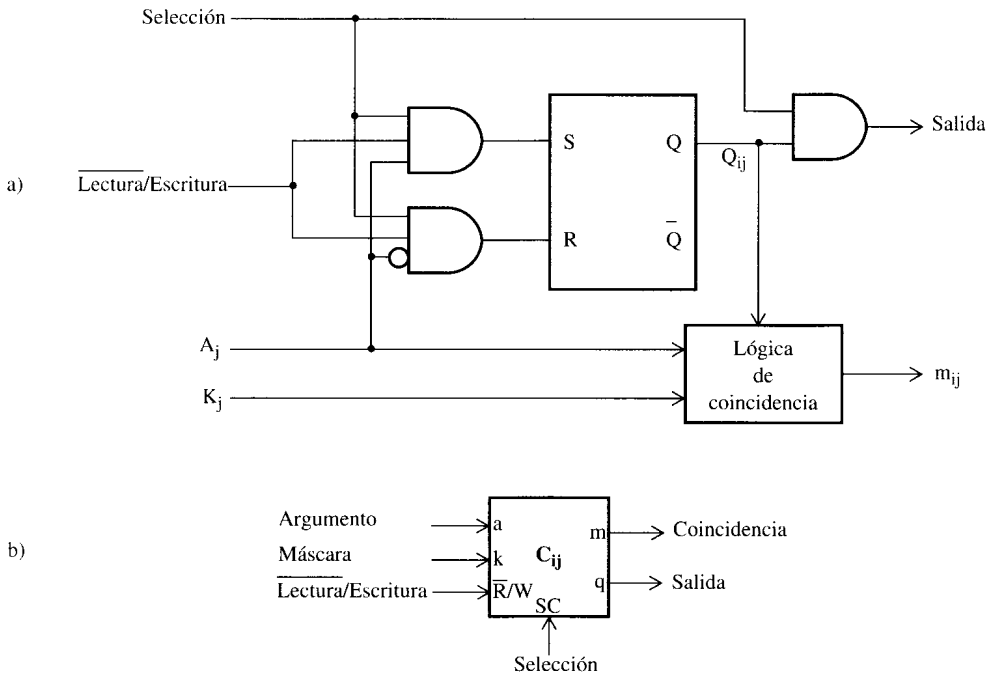


Figura 2.52: a) Celda básica de una memoria asociativa b) Esquema de la celda básica

a) La celda se selecciona con la señal *Selección* = 1 cada vez que se quiere leer o actualizar su contenido

b) Leer el bit almacenado en la celda durante una operación de lectura (*Selección* = 1, $\overline{\textit{Lectura/Escritura}}$ = 0)

c) Transferir al elemento de memoria el bit A_j del registro argumento durante una operación de escritura (*Selección* = 1, $\overline{\textit{Lectura/Escritura}}$ = 1)

d) Comparar el contenido de la celda con el correspondiente bit sin enmascarar del registro argumento, y proporcionar una salida m_{ij} para la lógica de decisión que actualiza el bit M_i del registro de marca

2.5.3 Determinación de la función lógica del registro de máscara

En la celda básica de memoria de la Figura 2.52 falta por calcular la función lógica del circuito de coincidencia entre los bits correspondientes de los registros argumento (A) y máscara (K). Se determina a partir del algoritmo de comparación de dos números binarios. Consta de dos etapas:

En la primera, no se consideran los bits del registro de máscara y se compara el contenido del registro A con los bits almacenados en las diferentes palabras de memoria. La palabra i-ésima es igual al contenido de A, si $A_j = Q_{ij}$, $j = 1, 2, ..., m$. La igualdad de dos bits se puede expresar mediante la siguiente función booleana:

$$m_{ij} = A_j Q_{ij} + \overline{A}_j \overline{Q}_{ij}$$

Para que una palabra i sea igual a A, se tiene que cumplir que $m_{ij} = 1$ para $j = 1, 2, ..., m$. Ésta es la condición para poner a 1 el bit M_i del registro de marca. La función booleana para esta condición es:

$$M_i = m_{i1}\, m_{i2}\, m_{i3} .... m_{im}$$

que corresponde a la operación AND de todos los bits que se comprueban en una palabra.

En la segunda etapa, se incluye el bit K_j del registro de máscara. El requisito es que si $K_j = 0 \Rightarrow A_j$ y Q_{ij} no necesitan comparación. Únicamente deben compararse cuando $K_j = 1$. Esto se consigue haciendo la operación OR de cada término con $\overline{K_j}$. Finalmente el bit M_i del registro de marca se puede expresar mediante la siguiente función booleana:

$$M_i = \prod_{j=1}^{k} (m_{ij} + \overline{K}_j) = \prod_{j=1}^{k} (A_i Q_{ij} + \overline{A}_j \overline{Q}_{ij} + \overline{K}_j)$$

donde $\prod$ representa el producto lógico de los m términos. Cada término producto en la expresión anterior será igual a 1 si se verifica que su correspondiente $K_j = 0$. Se necesitan n de estas funciones, una para cada palabra $i = 1, 2, ..., n$.

En la Figura 2.53 se muestra el circuito de la lógica de coincidencia para una palabra de la memoria asociativa. Cada celda requiere dos puertas AND y una puerta OR. Para A_j y K_j ($j = 1 .... m$) sólo se necesita un inversor por columna. La salida de todas las puertas OR en las celdas de la misma palabra van a la entrada de una puerta AND común para generar la señal M_i.

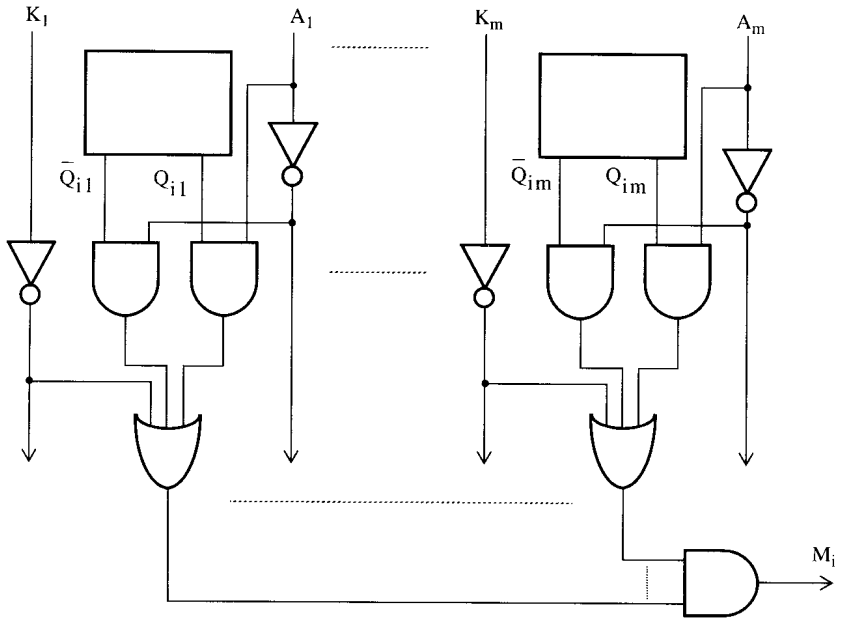


Figura 2.53: Estructura de una palabra de una memoria asociativa

M_i es igual a uno si se produce coincidencia y cero en caso contrario. Si el registro de máscara contiene sólo ceros, la salida de M_i valdrá 1 independiente del valor de A o de la palabra que hay en la memoria. En el funcionamiento normal de la memoria debe evitarse esta situación.

2.5.4 Operación de lectura

Si hay más de una palabra de la memoria que concuerda con los bits sin enmascarar del registro A, todas ellas tendrán un 1 en la posición correspondiente del registro de marca. Para la operación de lectura es necesario examinar uno a uno los bits del registro de marca. Si se conectan las salidas M_i del registro de marca M directamente a las líneas de selección de la palabra correspondiente (con la señal $\overline{R/W} = 0$) se tendrá de forma automática en las líneas de salida de la memoria asociativa el contenido de la palabra que coincide.

En la mayoría de las aplicaciones la memoria asociativa almacena una tabla que no tiene, para una máscara dada, dos filas idénticas. En este caso, solamente puede coincidir una palabra con los bits sin enmascarar del registro A. Aún más, si se excluyen de la memoria las palabras cuyo contenido sea cero, una salida de todos ceros indicará que no se ha producido coincidencia y que en consecuencia el elemento buscado no está disponible en la memoria.

2.5.5 Operación de escritura

Una memoria asociativa debe poder almacenar la información que se desee. Dependiendo de la aplicación, la escritura en este tipo de memorias puede adoptar diferentes formas .

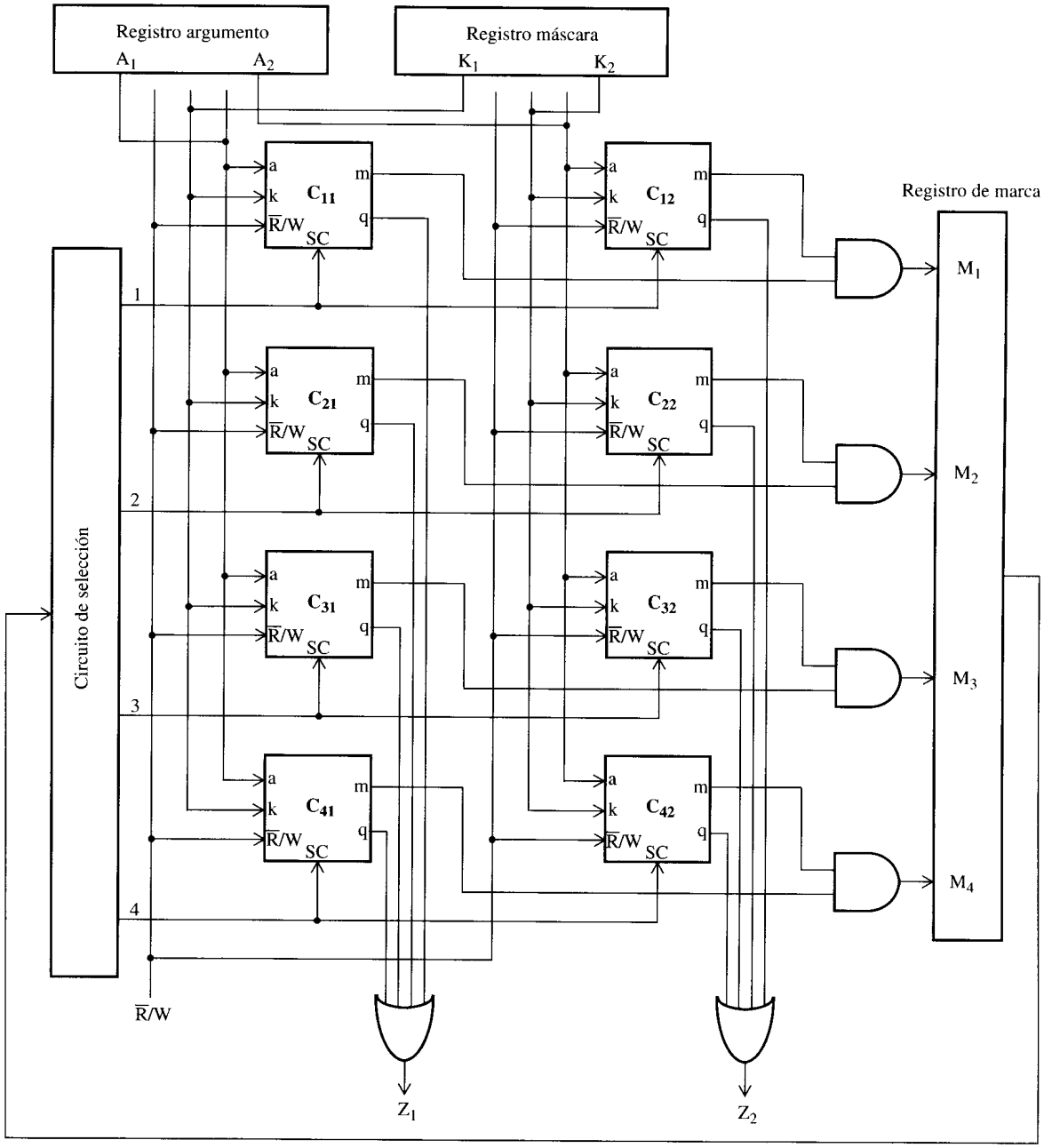


Figura 2.54: Diagrama de bloques de un array de memoria asociativa 4 × 2

Si previamente a las operaciones de búsqueda se carga la memoria con nueva información, su escritura se puede realizar direccionando secuencialmente cada posición. Esto convierte al dispositivo en una

memoria de acceso aleatorio para su escritura y en una memoria direccionable por contenido para su lectura. La ventaja en este caso es que la dirección para la entrada se puede decodificar como si fuese una memoria de acceso aleatorio. Así, en lugar de tener un bus de dirección de n líneas (una por cada palabra en memoria), se puede reducir por el decodificador a p líneas, donde $n = 2^p$.

Si se tienen que borrar las palabras que no se necesitan e insertar nuevas palabras una a una, es necesario un registro especial para distinguir entre *palabras activas e inactivas*. Este registro, denominado *registro etiqueta*, tendrá tantos bits como palabras hay en la memoria. Para cada palabra activa (es decir, que no se desea eliminar) almacenada en la memoria, el bit correspondiente del registro etiqueta se pone a 1. Una palabra se borra de memoria poniendo a 0 su bit del registro etiqueta. Las palabras se almacenan en memoria examinando el registro etiqueta hasta que se encuentra el primer bit igual a 0; esto indica la primera palabra inactiva y una posición para escribir una nueva palabra. Cuando se almacena en memoria la nueva palabra, se transforma en palabra activa y se pone a 1 su bit en el registro etiqueta. Cuando se suprime de memoria una palabra que no se necesita, se puede poner a 0 su bit correspondiente en el registro etiqueta. Las palabras con su bit del registro etiqueta a 0 deben enmascararse con los registros A y K, de forma que sólo se comparen palabras activas.

2.5.6 Ejemplo: Diseño de una memoria asociativa

Diseñar una memoria asociativa 4 × 2. En la Figura 2.54 se muestra el diagrama de bloques de la memoria asociativa de 4 palabras con 2 bits por palabra. Cuando se selecciona la operación de lectura $\overline{R}/W = 0$, se compara el contenido del registro argumento con las palabras cuyas líneas de selección están a 1. Las salidas de coincidencia de cada celda (m_{ij}) se aplican por filas a una puerta AND para generar así el bit de marca M_i para esa palabra. Si la operación es de escritura $\overline{R}/W = 1$, los bits del registro argumento se almacenan en la palabra seleccionada. ♦

Sección 2-6

Memorias compartidas

Hay ocasiones en las que el diseño de un sistema plantea la necesidad de que diferentes elementos tengan acceso a una misma unidad de memoria. Ejemplos característicos de esta situación son los accesos directos a memoria o el diseño de sistemas multiprocesadores.

En esta sección se examinan brevemente las técnicas fundamentales para compartir la utilización de una misma unidad de memoria entre varios elementos. En la Figura 2.55 se muestra la estructura general de un sistema de memoria compartida.

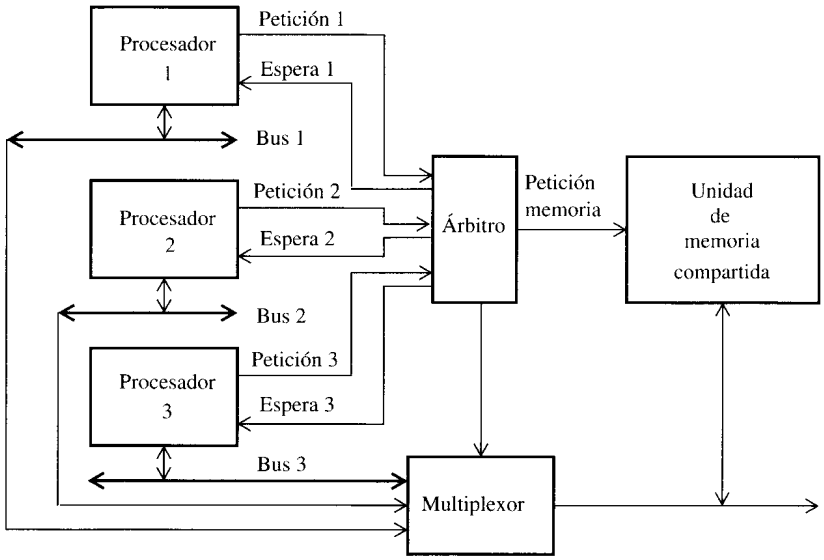


Figura 2.55: Diagrama de bloques de un sistema de memoria compartida

La unidad básica de estos sistemas es el *árbitro*, que es el elemento encargado de permitir el acceso a la unidad de memoria, en un instante dado, a cada uno de los procesadores que solicitan dicho recurso.

El árbitro se diseña de forma que asigne un tiempo de servicio, en promedio, análogo a todas las unidades que solicitan el recurso. Así, por ejemplo, en un conflicto entre los procesadores 1 y 2, el árbitro podría autorizar el acceso a la memoria al procesador 1 y la siguiente vez que suceda el mismo problema el acceso sería dado al procesador 2. Este criterio supone lo siguiente:

a) El árbitro tiene que guardar internamente información sobre las prioridades de servicio en caso de conflicto.

b) Esta prioridad se modifica cada vez que el árbitro atiende una petición de servicio.

El problema que se plantea es establecer la estrategia de asignación de prioridades a la petición de servicios de los distintos elementos. Los dos procedimientos más utilizados son:

- Asignación de la menor prioridad al elemento servido.
- Rotación de prioridades.

2.6.1 Asignación de la menor prioridad al elemento servido

La Tabla 2.14 muestra esta forma de reasignar las prioridades en el caso de tres procesadores que comparten una misma memoria.

En cualquier fila de la Tabla 2.14 la primera columna indica el estado presente de prioridades del árbitro, es decir, el orden actual de prioridades en el servicio a los tres procesadores que comparten la memoria. Las restantes columnas muestran, en función de las peticiones de servicio existentes, el procesador al que se le asigna el recurso y el nuevo orden de prioridades que reasigna el árbitro.

Para cualquier estado presente, la transición al próximo estado se obtiene asignando la menor prioridad al elemento servido. Resulta inmediato ver que con esta estrategia de cálculo de prioridades, la complejidad del árbitro crece de forma factorial con el número de procesadores a servir.

Así, en el caso de tres procesadores el árbitro tendrá 6 estados, y para seis procesadores se pasarán a 720 estados. Esta explosión en la complejidad del árbitro hace poco aconsejable la implantación de este método de reasignación de prioridades cuando el número de procesadores es elevado.

Estado presente	Peticiones simultáneas de servicio						
	1	2	3	1, 2	1, 3	2, 3	1, 2, 3
1-2-3	1 2-3-1	2 1-3-2	3 1-2-3	1 2-3-1	1 2-3-1	2 1-3-2	1 ← elemento servido 2-3-1 ← nuevo orden de prioridades
1-3-2	1 3-2-1	2 1-3-2	3 1-2-3	1 3-2-1	1 3-2-1	3 1-2-3	1 3-2-1
2-1-3	1 2-3-1	2 1-3-2	3 2-1-3	2 1-3-2	1 2-3-1	2 1-3-2	2 1-3-2
2-3-1	1 2-3-1	2 3-1-2	3 2-1-3	2 3-1-2	3 2-1-3	2 3-1-2	2 3-1-2
3-1-2	1 3-2-1	2 3-1-2	3 1-2-3	1 3-2-1	3 1-2-3	3 1-2-3	3 1-2-3
3-2-1	1 3-2-1	2 3-1-2	3 2-1-3	2 3-1-2	3 2-1-3	3 2-1-3	3 2-1-3

Tabla 2.14: Asignación de la menor prioridad al elemento servido

2.6.2 Rotación de prioridades

Con este procedimiento se intenta reducir el número de estados del árbitro y en consecuencia simplificar su diseño. En un estado cualquiera, el próximo estado se calcula rotando el orden de prioridades actual hasta que el procesador al que se acaba de dar servicio tiene la menor prioridad.

En la Tabla 2.15 se dan las transiciones de estados con esta estrategia para el mismo caso de la tabla anterior. Es fácil verificar que ahora el número de estados del árbitro coincide con el número de elementos a servir. Sin embargo, el reparto de los tiempos de servicio ya no es tan óptimo como en el caso previo.

La razón es que ahora, en general, no tiene por qué cumplirse que disminuya sólo la prioridad del procesador al que se acaba de dar servicio. Véase por ejemplo la casilla marcada con asterisco en la Tabla 2.15. (Al pasar del estado presente (1-2-3) al próximo estado (3-1-2), además de reducirse la prioridad del elemento 2, que es el procesador servido, disminuye también la del elemento 1).

Estado presente	**Peticiones simultáneas de servicio**						
	1	2	3	1, 2	1, 3	2, 3	1, 2, 3
1-2-3	1 2-3-1	2 3-1-2	3 1-2-3	1 2-3-1	1 2-3-1	(*) 2 3-1-2	1 ← elemento servido 2-3-1 ← nuevo orden de prioridades
2-3-1	1 2-3-1	2 3-1-2	3 1-2-3	2 3-1-2	3 1-2-3	2 3-1-2	2 3-1-2
3-1-2	1 2-3-1	2 3-1-2	3 1-2-3	1 2-3-1	3 1-2-3	3 1-2-3	3 1-2-3

Tabla 2.15: Asignación de la menor prioridad mediante la rotación de prioridades

Sección 2-7

Memorias tipo pila

Estas memorias devuelven la información de modo inverso al de su almacenamiento, y por lo tanto, se caracterizan por el hecho de que sólo se puede leer el último dato almacenado en ellas. También se las conoce por el nombre de *memorias LIFO* (Last In First Out). Desde el punto de vista del acceso a los datos funcionan como memorias de acceso secuencial.. A diferencia de una memoria RAM en la que cualquier dato almacenado puede ser accedido en cualquier momento, a los datos almacenados en una pila sólo se puede acceder a través de una posición concreta. Se pueden ver así como un componente de memoria con un acceso limitado.

Únicamente se puede extraer el dato almacenado en la *cumbre de la pila.* Cuando se inserta un dato en la pila sustituye al que estaba en la cumbre y toda la información almacenada en la pila desciende un lugar hacia abajo. De forma recíproca, cuando se extrae de una pila (el dato extraído es el que ocupa la cumbre de la pila) toda la información almacenada en la pila asciende un lugar hacia arriba (ver Figura 2.56). Cuando se diseña una pila de inserción/extracción, desde este punto de vista, la cuestión importante a considerar es que los datos almacenados se desplazan en una posición descendente o ascendente en las operaciones de inserción y extracción respectivamente. Para su implementación hardware se pueden utilizar tanto registros de desplazamiento como un contador reversible.

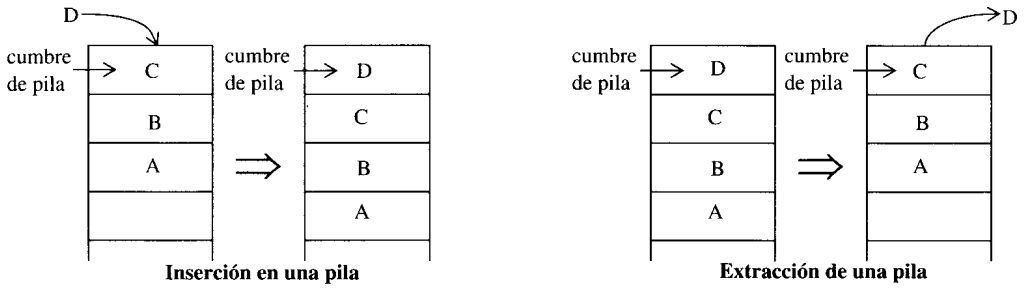


Figura 2.56: Operaciones de inserción/extracción en una pila implementada con registros de desplazamiento

La principal desventaja de la realización con registros de desplazamiento es el gran número de ellos que se necesitan cuando se diseña una pila de gran capacidad, lo que encarece su coste. Por esta razón una forma más efectiva de ver las operaciones de inserción/extracción en una pila es la que se muestra en la Figura 2.57 en la que los datos en la pila no modifican sus posiciones y lo que se hace es incrementar o decrementar el valor de la cumbre de pila. La implementación ahora es posible hacerla empleando una memoria RAM. Como las memorias RAM no son capaces de desplazar su contenido, las operaciones de inserción y extracción en estas pilas se realizan de una manera algo diferente: cambiando la posición de la cumbre de la

pila. En otras palabras, al insertar datos en la pila, la dirección de la cumbre se incrementará por cada inserción y por el contrario se decrementará en cada extracción.

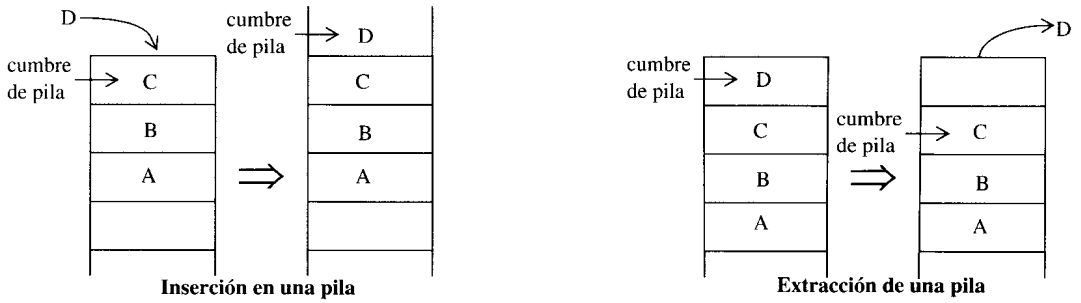


Figura 2.57: Operaciones de inserción/extracción en una pila que modifica el valor del puntero cumbre de pila

Se analizan a continuación estas dos formas de realizar una memoria tipo pila:

a) *Mediante un conjunto de registros convenientemente interconectados.*

Es la organización usual de una pila cuando ésta forma parte de la propia CPU (ver Figura 2.58). Cada uno de los registros que constituyen la pila puede copiarse sobre los registros adyacentes. Desde el exterior sólo se pueden hacer operaciones de inserción/extracción sobre el registro R_1 (cumbre de la pila).

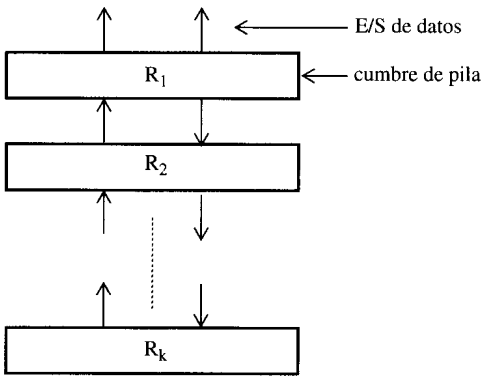


Figura 2.58: Registros interconectados como memoria pila

En la Figura 2.59 se muestra la realización de una pila con k registros de 1 bit/registro. Cuando la señal $\overline{R}/W = 0$ se está en modo de lectura o inserción y el dato de entrada D_e se carga en el elemento de memoria 1 (que es la cumbre de la pila) y el contenido del elemento de memoria j se transfiere al elemento de memoria $j + 1$ ($j = 1, 2, ..., k$), con lo que la información desciende a lo largo del registro de desplazamiento. Cuando la señal $\overline{R}/W = 1$ se está en modo de escritura o extracción y el contenido del elemento de memoria 1 se transfiere como dato de salida D_s y el contenido del elemento de memoria $j+1$ se transfiere al elemento de

memoria j (j = 2, ..., k), con lo que la información asciende a lo largo del registro de desplazamiento.

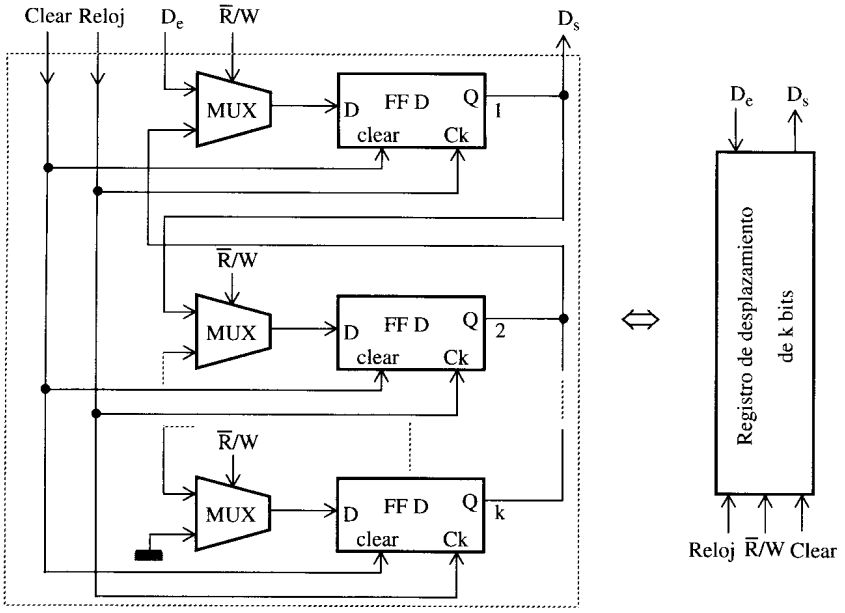


Figura 2.59: Pila con k registros y 1 bit/registro

Se puede construir fácilmente una pila con k registros de m bits/registro interconectando adecuadamente m pilas del tipo anterior (ver Figura 2.60).

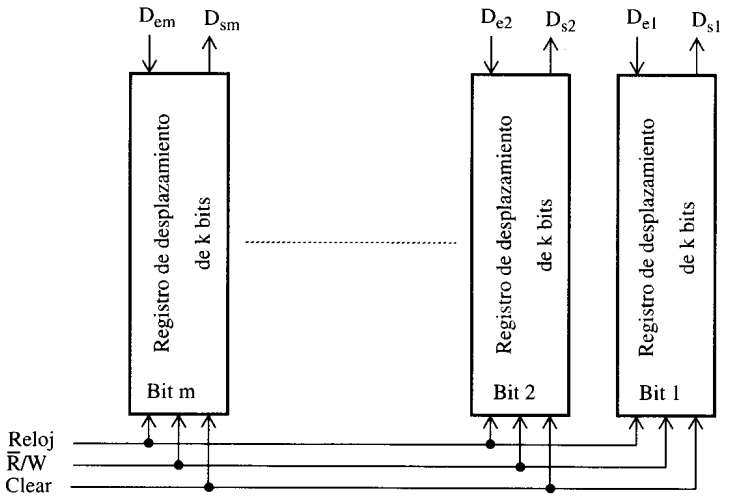


Figura 2.60: Pila con k registros y m bits/registro

b) *Mediante un conjunto de palabras de la memoria principal.*

Para efectuar esta realización de la pila, el procesador dispone de un registro especial que se conoce como *puntero de pila* o *registro SP* (Stack Pointer). El procesador posee también un repertorio de instrucciones que le permiten realizar las operaciones básicas de inserción y extracción de un dato en la pila .

El contenido del puntero de pila especifica siempre la posición de memoria donde se ha de almacenar el próximo dato que se introduzca en la pila (ver Figura 2.61).

En consecuencia, será necesario para la realización de una pila en memoria, que la CPU disponga de instrucciones que de forma automática incremente o decremente el contenido del registro *SP*, cada vez que se realiza un acceso a la pila.

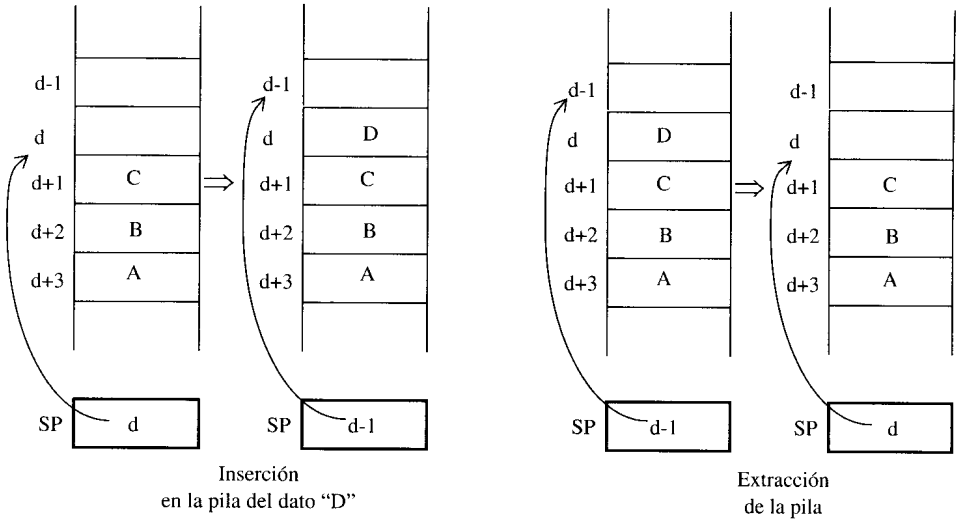


Figura 2.61: Zona de la memoria principal utilizada como pila

En la Tabla 2.16 se resumen las ventajas y desventajas de las dos formas de realizar una pila.

	Pila realizada como conjunto de registros	**Pila formada por una zona de la memoria principal**
Ventajas	• Velocidad alta	• Utiliza la memoria principal • Coste bajo • Gran capacidad
Desventajas	• Hardware propio en la CPU • Coste alto • Capacidad limitada	• Velocidad baja

Tabla 2.16: Comparación de las dos formas de realizar una pila

A veces se suelen dar realizaciones de carácter mixto, en las que las primeras posiciones de la pila están soportadas por un conjunto de registros del procesador y el resto de la información sobre posiciones de memoria.

2.7.1 Aplicaciones de las memorias tipo pila

Entre las aplicaciones más clásicas de este tipo de memorias se pueden mencionar las siguientes:

a) En el salto a subrutinas, para almacenar la dirección de retorno. El empleo de una pila facilita el tratamiento de los saltos anidados.

En la Figura 2.62 se muestra de forma esquemática la utilización de una memoria tipo pila en la construcción de instrucciones de salto a subrutina con varios saltos concatenados, en cuyo caso se utiliza la pila para almacenar las direcciones de ruptura de secuencia.

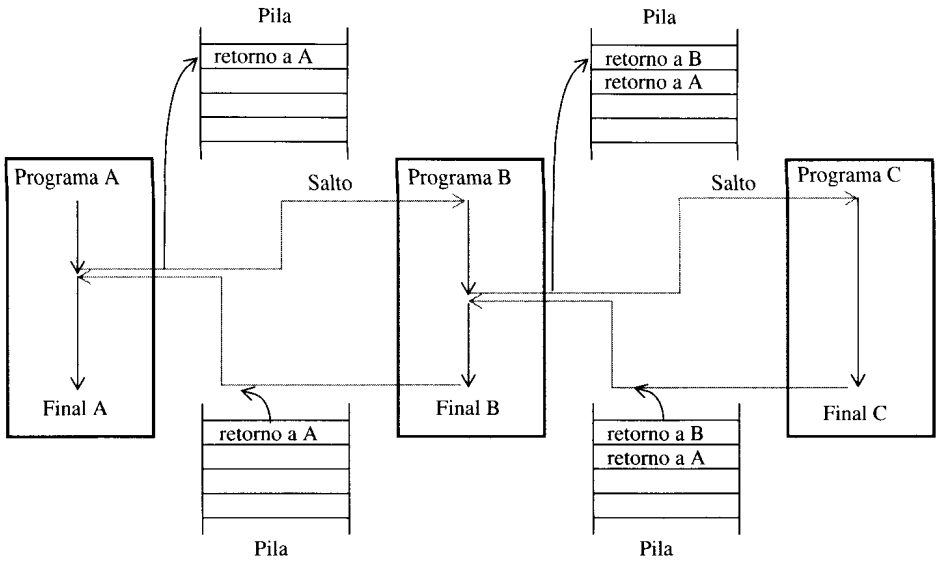


Figura 2.62: Utilización de una pila en el salto a subprogramas

b) En el tratamiento de interrupciones, para guardar el estado de los registros del procesador antes de la interrupción. También simplifica el tratamiento de interrupciones anidadas (ver apartado 3-4.4).

c) En la compilación de expresiones aritméticas.

El siguiente ejemplo ilustra la evaluación de la expresión $(A + B) \times (C - D)$ en un computador hipotético que se basa en la utilización de una pila. Se supone que la instrucción *PUSH* inserta el contenido del registro *D*0 en la pila. Las instrucciones *SUM, RES* y *MUL* todas ellas extraen los dos operandos que necesitan de la parte superior de la pila, realizan la operación indicada e insertan el resultado otra vez en la cumbre de la pila. Finalmente *POP* extrae el contenido de la posición de memoria referenciada por la cumbre de la pila y lo almacena en el registro *D*0.

1	MOVE A, D0	Copia el contenido de A en D0
2	PUSH	Inserta el contenido de D0 en la pila
3	MOVE B, D0	Inserta el contenido de B en D0
4	PUSH	Inserta el contenido de D0 en la pila
5	ADD	Suma los dos operandos en cumbre y deja resultado en cumbre de pila
6	MOVE C, D0	Copia el contenido de C en D0
7	PUSH	Inserta el contenido de D0 en la pila
8	MOVE D, D0	Copia el contenido de D en D0
9	PUSH	Inserta el contenido de D0 en la pila
10	SUB	Resta los dos operandos en cumbre y deja resultado en cumbre de pila
11	MUL	Multiplica los dos operandos en cumbre y deja resultado en cumbre de pila
12	POP	Extrae el resultado de la cumbre de pila y lo almacena en D0

La Figura 2.63 representa el estado de la pila en diferentes etapas de la ejecución del segmento de programa anterior. Los números debajo de cada diagrama se corresponden con el número de línea del programa.

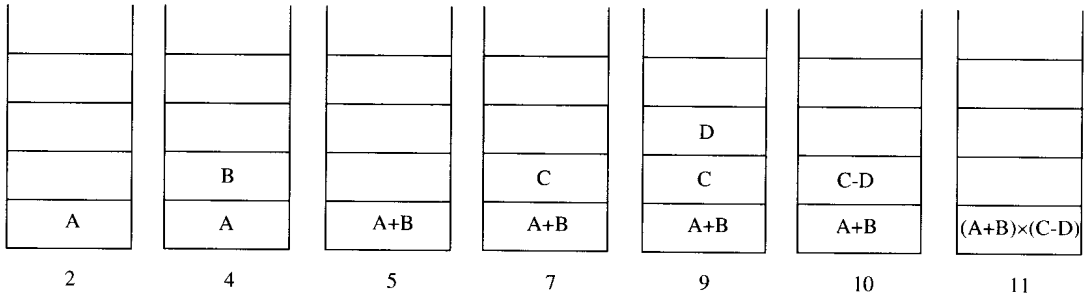


Figura 2.63: Estado de la pila en la evaluación de la expresión (A + B) × (C - D)

Sección 2-8

Discos magnéticos

Los discos son el principal soporte (y en muchos sistemas el único) para la memoria secundaria y el sistema de archivos, por ello en esta sección se estudiarán las características funcionales y la organización de la información en estos periféricos.

2.8.1 Estructura física

El medio físico de almacenamiento de los datos es una película de óxido magnético que recubre a un disco que se parece a los de grabación fonográfica. Estos platos de grabación pueden ser *extraibles* o *fijos*, dependiendo de si pueden ser extraidos o no de la unidad. Una unidad también puede disponer de varios platos. En la Figura 2.64 se muestra la configuración mecánica básica de una unidad de disco.

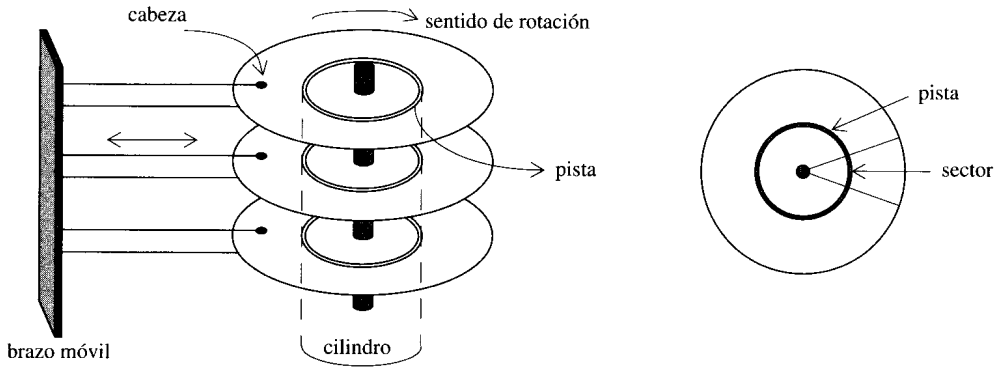


Figura 2.64: Configuración mecánica de una unidad de disco

Sea el disco extraible o fijo, una vez situado en la unidad el funcionamiento es similar. Los discos giran a una velocidad constante (superior a las 3.000 rpm, aunque la velocidad de giro de los discos extraibles es menor). El número de discos varía entre 1 (para los disquetes) y 20 en los discos de gran capacidad. Los datos se leen y escriben mediante *cabezas de lectura/escritura* montadas de forma que contacten con la parte del disco que contiene los datos. Cada disco tiene dos *superficies* (o caras) por lo que hay dos cabezas de lectura y escritura para cada disco. Los datos se almacenan en las superficies magnéticas del disco en forma de círculos concéntricos llamados *pistas*. Se llama *cilindro* al conjunto de pistas de todas las superficies que están a la misma distancia del eje del disco. Las pistas se dividen en *sectores* y cada sector contiene varios centenares de bytes.

Conviene advertir que, como la velocidad de giro, el número de sectores por pista y el número de bytes por sector son constantes pero las pistas tienen distinto radio, la densidad de grabación en las pistas internas será mayor que en las externas. Es posible optimizar la superficie magnética del disco (aumentando así su capacidad) si el número de sectores por pista es menor cuanto más interna sea ésta. Normalmente no se considera esta posibilidad ya que complica el diseño del controlador asociado y de su software. Sin embargo, es previsible que en el futuro inmediato comiencen a aparecer en el mercado unidades de discos con estas características cuando se generalice el uso de controladores basados en la norma *SCSI*.

Dependiendo del número de cabezas, los discos pueden ser de *cabeza fija* o de *cabeza móvil*. Los de cabeza fija tienen generalmente una cabeza de lectura/escritura por cada pista, de forma que a un sector se accede activando la cabeza sobre la pista apropiada cuando el sector pasa bajo ella. El tiempo necesario para acceder al sector deseado se denomina *latencia rotacional* (o tiempo de espera) y en promedio es igual a la mitad del tiempo de revolución del disco. La velocidad típica de giro de los discos suele ser de 3600 rpm que equivale a dar una revolución cada 16,7 mseg. Así el retardo medio rotacional será de 8,3 mseg. Los disquetes giran sin embargo mucho más lentamente, entre 300 y 600 rpm, por lo que el tiempo medio de latencia estará comprendido entre 100 y 200 mseg.

Los discos de cabeza móvil tienen una cabeza o unas pocas cabezas de lectura/escritura por superficie. En este caso, la lectura de un sector requiere que el montaje de las cabezas sea desplazado hasta el cilindro correspondiente, tras de lo cual se activa la cabeza cuando el sector deseado pasa bajo ella. En este caso el *tiempo de acceso* (t_a) está compuesto por el *tiempo de posicionamiento de la cabeza o tiempo de búsqueda* (t_b) y el *tiempo de latencia rotacional* (t_r). Debido a estas dos componentes en el tiempo de acceso hay una gran variabilidad de unos discos a otros. Pero una vez alcanzado el sector, la velocidad de transferencia de datos de los discos actuales suele ser del orden de 5 Mb/seg. Debido a estas velocidades elevadas, lo normal es usar técnicas de acceso por DMA para la transferencia de datos entre los discos y la memoria principal. En la Tabla 2.17 se muestran el rango de los parámetros de los discos actuales.

Número de superficies	$2 \leq S \leq 40$
Número de cilindros	$40 \leq C \leq 2000$
Número de sectores por pista	$10 \leq S \leq 100$
Número de bytes por sector	256, 512 o 1024
Capacidad del disco	360 KB $\leq M \leq$ 10 GB
Tasa de transferencia	60 KB/s $\leq N \leq$ 4 MB/s
Tiempo medio de búsqueda	1 mseg $\leq t_{bm} \leq$ 50 mseg
Tiempo de búsqueda (cilindros contiguos)	0.1 mseg $\leq t_{bc} \leq$ 5mseg
Tiempo medio de espera	10 mseg $\leq t_e \leq$ 200 mseg
Tiempo de transferencia de un sector	0.1 mseg $\leq t_t \leq$ 20 mseg
Tiempo de encendido/apagado del motor	10 mseg $\leq t_e \leq$ 200 mseg

Tabla 2.17: Parámetros típicos de los discos

En resumen, el tiempo que se precisa para que los datos se transfieran desde el disco a la memoria principal del computador se compone de los tres tiempos siguientes (ver Figura 2.65):

1) *Tiempo de búsqueda* (t_b): Es el tiempo necesario para que las cabezas se desplacen al cilindro adecuado. Consta de dos componentes claves: el tiempo de arranque inicial (t_i) y el tiempo que se tarda en recorrer todos los cilindros que hay entre la pista inicial y la final. Desgraciadamente no es una función lineal del número de pistas que tiene el disco. Se suele aproximar con la fórmula siguiente:

$$t_b = m \times n + t_i$$

donde

m = cte que depende de la unidad del disco

en un disco con tecnología Winchester[1] de bajo coste para computadores personales $m \cong 0.3$ mseg y $t_i \cong 20$ mseg, mientras que para discos más caros son $m \cong 0.1$ mseg y $t_i \cong 3$ mseg

2) *Retardo rotacional* (t_r): Es el tiempo medio que tarda el sector en estar debajo de la cabeza de lectura/escritura. Si f es la velocidad de rotación en revoluciones por segundo, t_r se puede expresar por:

$$t_r = \frac{1}{2f}$$

3) *Tiempo de transferencia* (t_t): Es el tiempo que se tarda en transferir los datos. Depende del tamaño del sector y es el menor de los tres. Si b es el número de bytes que se desean transferir y P el número de bytes que hay en una pista, t_t se puede expresar como:

$$t_t = \frac{b}{P \times f}$$

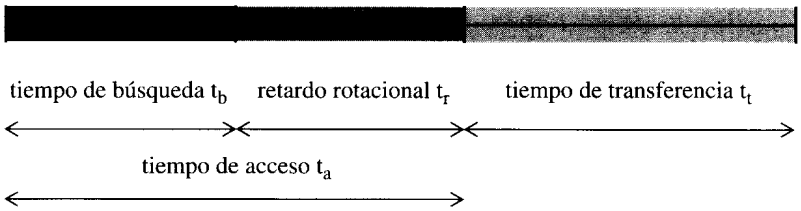


Figura 2.65: Temporización de la transferencia de E/S del disco

1. En la mayoría de las unidades de "disco duro" el disco no está accesible ya que se encuentra en un compartimento estanco. Esta tecnología se conoce como "tecnología Winchester". Su ventaja está en aislar a la unidad de las particulas de polvo del aire lo que permite que las cabezas estén más próximas a la superficie magnética y que las pistas sean más estrechas, aumentando así su capacidad y reduciendo su tamaño y su peso. El nombre proviene de la unidad de disco de *IBM 3030* que almacenaba 30 MB en cada lado de una única superficie. El número del modelo es una reminiscencia del famoso rifle Winchester 3030.

En algunos supercomputadores, con el fin de mejorar el rendimiento de la transferencia de datos entre el disco y la memoria principal, se suele emplear una técnica conocida como *sensor de posición rotacional*. La forma en la que opera es la siguiente: cuando se emite una orden de "búsqueda de un sector" se libera el canal para manejar las operaciones de E/S. Al finalizar esta orden, el dispositivo determina en que momento el sector pasará por encima de la cabeza y cuando se va aproximando intenta restablecer de nuevo la comunicación. En el caso de que la unidad de control del computador o el canal estén ocupados atendiendo a otra operación de E/S fallará el intento de reconexión y el dispositivo deberá girar una vuelta completa antes de volver a intentarlo otra vez.

2.8.2 Ejemplo: Tiempo de acceso a un archivo de acceso secuencial y aleatorio

Es preciso tener presente que uno de los factores dominantes a la hora de medir la velocidad de las operaciones de E/S entre el disco y la memoria principal es la forma como está organizado el archivo que se va a transferir. Para poner esto de manifiesto se supone que se desea leer un archivo que tiene un tamaño de *256K*. El disco utilizado tiene los siguientes parámetros:

tiempo medio de búsqueda = 25 mseg

velocidad de transferencia = 800 Kb/s

tamaño del sector = 256 bytes

número de sectores por pista = 64

Se van a considerar dos situaciones extremas en cuanto a la organización del archivo en el disco

1) *Acceso secuencial.* El archivo está almacenado de la forma más compacta posible ocupando 1024 sectores localizados en 16 pistas contiguas

2) *Acceso aleatorio.* El archivo está almacenado de forma que los sectores están distribuidos aleatoriamente por su superficie

En la Figura 2.66 se muestra el tiempo total empleado en la lectura del archivo en los dos casos. En el acceso secuencial se ha supuesto que una vez leída la primera pista, en las restantes el tiempo de búsqueda es despreciable al estar ya posicionada correctamente la cabeza de lectura. Resulta claro, de este ejemplo, que el orden en el que se leen los sectores del disco tiene un efecto muy importante sobre su rendimiento. ♦

2.8.3 Controlador del disco

Las señales típicas entre el controlador del disco y la unidad propiamente dicha están indicadas en la Figura 2.67. Puesto que un controlador puede manejar varias unidades de características similares, se necesitan algunas líneas de control para seleccionar la unidad, son las marcadas como *líneas de selección de unidad.* Análogamente, las *líneas de selección de cabeza* se utilizan para activar la cabeza de lectura/escritura de la unidad adecuada. Después vienen las *líneas de dirección, paso, leer y escribir*, la primera de ellas indicarán como se desplazará la cabeza desde la situación actual (hacia dentro o hacia fuera), la segunda proporciona una secuencia de impulsos, y la tercera y cuarta se usa para activar la cabeza seleccionada. Las *líneas de entrada y salida de datos* transportan los bits leídos del disco o los que hay que grabar en el mismo según sea la operación de lectura o de escritura. Después vienen una serie de entradas como las de *reinicializar, indicación de fallo*, etc.

a) Acceso secuencial

Tiempo para leer la 1ª pista

Tiempo medio de búsqueda	25,0	mseg
Retardo rotacional	8,3	mseg
Tiempo de lectura de 64 sectores	20,5	mseg
	53.8	mseg

Tiempo para leer cada pista sucesiva

Retardo rotacional	8,3	mseg
Tiempo de lectura de 64 sectores	20,5	mseg
	28,8	mseg

Tiempo total = 53,8 + 15 × 28,8 = 485,8 mseg

↑ número de pistas que quedan por leer

b) Acceso aleatorio

Tiempo para leer cada sector

Tiempo medio de búsqueda	25,0	mseg
Retardo rotacional	8,3	mseg
Tiempo de lectura de 1 sector	0,32	mseg
	33,62	mseg

Tiempo total = 1024 × 33.62 = 34427 mseg

↑ número de sectores que hay que leer

Figura 2.66: Lectura de un archivo mediante acceso secuencial y acceso aleatorio

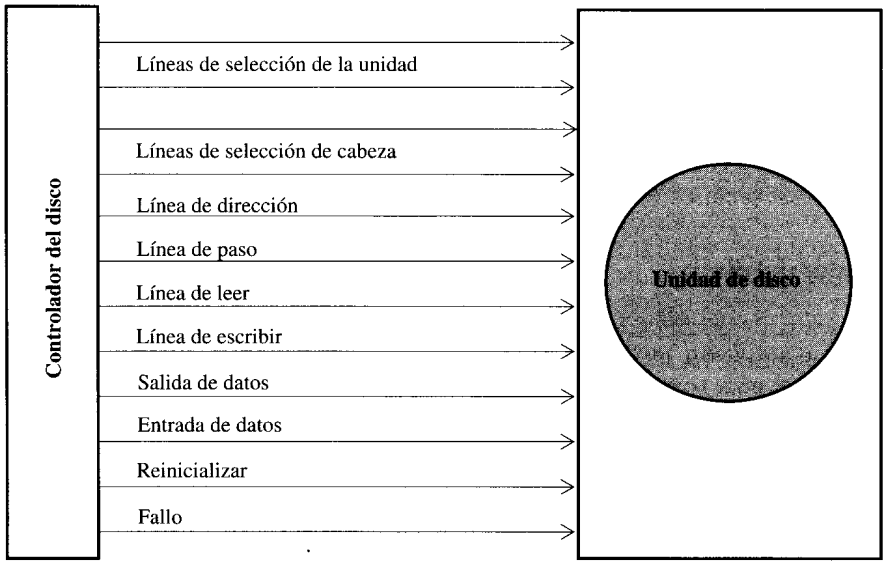


Figura 2.67: Señales entre el controlador y la unidad de disco

Las funciones y estructura del controlador del disco se verá en el tema 3 cuando se estudien los controladores de E/S. El control de errores es necesario para detectar los errores transitorios provocados por ruidos e interferencias electromagnéticas y para manejar los defectos del medio de almacenamiento. Cada vez que se escribe un dato en el disco, el controlador calcula los bits de verificación y los añade a la información que se transmite. Como los errores de disco ocurren en secuencias y producen errores en cadenas de bits sucesivos, los controladores suelen utilizar mecanismos de detección capaces de tratarlos, como los códigos de redundancia crítica. Dependiendo de la causa, un error puede ser transitorio o permanente. Los primeros se corrigen leyendo varias veces el sector correspondiente. Si después de unas cuantas lecturas (que dependen de un parámetro denominado contador de reintentos) el error persiste, éste se trata como permanente. En este caso el bloque queda marcado como defectuoso, de forma que se impide su utilización. La misión de evitar el uso de los bloques defectuosos corresponde generalmente a una capa de bajo nivel del sistema de control de archivos. Los bloques defectuosos pueden ser detectados durante la fase de inicialización del disco.

Otros errores, denominados errores de búsqueda, pueden deberse a problemas mecánicos en las cabezas. Un ejemplo de esta clase puede ser que el brazo de la cabeza de lectura/escritura no se posiciona correctamente. Algunos controladores corrigen automáticamente los errores de búsqueda, pero otros sólo activan un bit de error y pasan el resto del problema al manejador, que se encarga del error enviando una orden de recalibración.

Un aspecto que conviene considerar es la necesidad de que el controlador disponga de un buffer interno para el almacenamiento temporal de los datos. La razón es que una vez se ha iniciado una operación de E/S desde el disco, los datos se transfieren a velocidad constante, independientemente de que el controlador esté preparado o no para recibirlos. Si el controlador pretende transferirlos directamente a la memoria es preciso que utilice el bus del sistema, que sin embargo podría estar ocupado por otro dispositivo. La utilización de un buffer interno hace innecesario el bus del sistema hasta que comienza el acceso por DMA (lo que facilita el diseño del controlador ya que la transferencia a memoria mediante DMA no impone limitaciones de tiempo) y tiene consecuencias importantes en la gestión del sistema de E/S. Cuando el controlador o la propia CPU están transfiriendo los datos a la memoria (operación de salida), el siguiente sector estará pasando por debajo de las cabezas de lectura de manera que la información correspondiente estará llegando al controlador (operación de entrada). Si éste tiene un diseño sencillo no permite realizar de forma simultánea la entrada y la salida de datos por lo que durante la transferencia a memoria no hay posibilidad de leer el siguiente sector.

Como resultado del argumento anterior se deduce que este tipo de controladores no es capaz de leer bloques contiguos de información (que están en sectores adyacentes). Así, la lectura de una pista completa requiere al menos dos rotaciones, una para los sectores pares y otra para los impares (ya que si el tiempo de transferencia de los bloques desde el controlador a la memoria fuese mayor que el empleado en leerlos desde el disco, podría ser necesario leer un bloque y seguidamente saltar dos o más). Esta técnica de saltar bloques, para dar un margen de tiempo suficiente para que el controlador transmita los dato a la memoria, se llama *entrelazado*. Por esta razón cuando se formatean los discos, los sectores se numeran teniendo en cuenta el *factor de entrelazado*. En la Figura 2.68 se muestra un disco con 16 sectores por pista formateado con distintos factores de entrelazado.

Hay pues una gran variedad de controladores de discos, aunque cada vez son más "inteligentes", encargándose de muchas de las funciones que anteriormente se hacían por software, como son el formateo o inicialización de los discos, el almacenamiento en buffers o la agrupación de los datos en bloques.

Las funciones que se tienen que hacer en un disco se pueden describir mediante una típica rutina de

disco, que básicamente permita la lectura y escritura de sectores especificados por medio de direcciones físicas de la forma siguiente:

<número de cilindro, número de cabeza, número de sector>

Las órdenes elementales al nivel de la rutina son *buscar*, *leer* y *escribir*. En algunas implementaciones las peticiones de leer y escribir tienen que venir precedidas por órdenes de búsqueda.

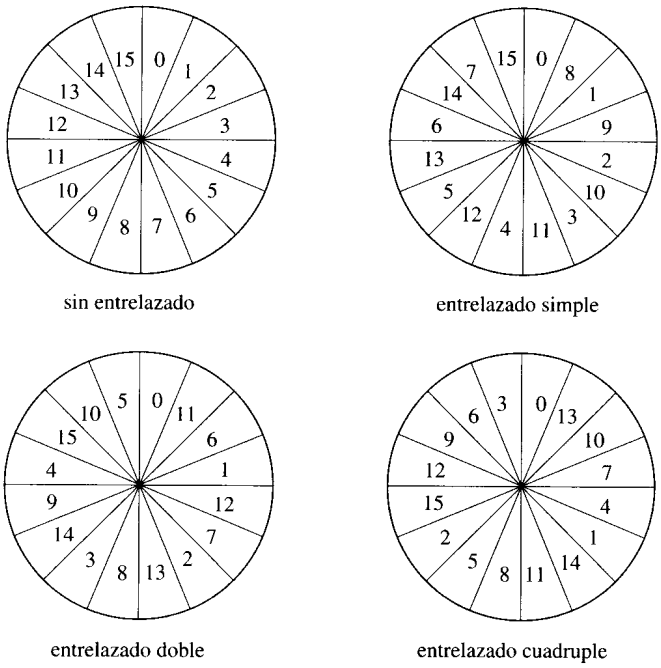


Figura 2.68: Entrelazado de bloques

Como las transferencias de datos entre el disco y la memoria se hace mediante DMA, la rutina de disco no devuelve datos explícitos a sus invocadores, excepto la información referente al resultado de la operación. Cuando se detecta una excepción, como por ejemplo un error de lectura o un fallo de la unidad, la rutina del disco indica el número real de bytes transferidos correctamente antes del error. Otras órdenes de la rutina pueden incluir algunos modos de manejo de la unidad de almacenamiento, como el de formatear una pista o marcar un bloque como defectuoso.

Algunas rutinas y controladores son capaces de transferir múltiples sectores o incluso pistas, en respuesta a una sola orden. Esto presenta la ventaja de que las cabezas sólo se tienen que posicionar una vez, lo que reduce el tiempo medio de acceso para muchos sectores. Estas operaciones requieren que los datos ocupen direcciones consecutivas de disco.

2.8.4 Planificación del disco

El tiempo de lectura/escritura de un sector del disco depende de tres factores: a) el tiempo de búsqueda del cilindro, b) el retardo rotacional y c) el tiempo de transferencia (ver Figura 2.65). De todos ellos, el único que

se puede optimizar desde el programa gestor del disco es el primero, ya que los otros dos dependen de las características propias del disco y del bus de transmisión. Esto se observa claramente si se analizan los resultados del Ejemplo 2.8.2. El motivo de la diferencia de rendimiento entre los accesos secuencial y aleatorio está en el tiempo de búsqueda. Si las peticiones de acceso a lo sectores del disco se realizan seleccionando las pistas de forma aleatoria, entonces la velocidad de las operaciones de E/S entre el disco y la memoria principal es muy baja. Una forma de mejorar los resultados consiste en reducir el tiempo medio empleado en las búsquedas. Cuando un programa requiere una operación de E/S del disco, envía la correspondiente llamada al sistema operativo especificándole las siguientes informaciones:

a) Tipo de operación (si se trata de una entrada o de una salida)

b) Dirección en el disco (unidad, cilindro, superficie, bloque)

c) Dirección en memoria

d) Cantidad de información que se va a transferir (número de bytes)

Si está disponible, es posible atender de forma inmediata la petición. No obstante, si la unidad o el controlador del disco se encuentran sirviendo una solicitud anterior, será preciso poner en una cola todas las peticiones que vayan llegando (generalmente provienen de otros programas). En los sistemas con multiprogramación, la mayoría de las veces la cola del disco no estará vacía, de manera que al acabar una petición habrá que elegir una de las que están pendientes de ser servidas.

Planificación FCFS: primero en llegar, primero en ser servido

Este método es la forma más sencilla de gestionar la búsqueda del sector. Se realiza directamente ya que las solicitudes de acceso se van almacenando en una memoria tipo FIFO, de forma que la primera petición que llega es la primera que se sirve; es el algoritmo FCFS (First Come, First Served). Este algoritmo es sencillo de programar y se puede considerar que es inherentemente justo, pero es muy probable que no ofrezca en promedio el mejor tiempo de servicio. Sea, por ejemplo, un disco con 200 pistas para el que se dispone de una cola ordenada en la que se han almacenado las siguientes peticiones de pistas:

22, 124, 105, 181, 142, 36, 5, 59, 115

la posición inicial de la cabeza de lectura/escritura está en la pista 95. En la Tabla 2.18 se resumen los resultados de la estrategia de planificación FCFS. En un principio la cabeza se moverá de la pista 95 a la 22, luego a la 124 y así sucesivamente, atendiendo por el orden de llegada todas las peticiones que se encuentran en la cola.

Próxima pista a la que se accede	22	124	105	181	142	36	5	59	115	
Número de pistas que se atraviesan	73	102	19	76	39	106	31	54	56	LMB = 61,8

Tabla 2.18: Algoritmo FCFS de planificación del disco

La longitud media de búsqueda (LMB) es de 61,8. El problema con el algoritmo FCFS está en los movimientos bruscos de vaivén a los que se ve sometida la cabeza de lectura/escritura (por ejemplo ir de la pista 142 a la 36). Si se pudieran atender juntas las peticiones de las pistas 22, 36, 5 y 59 antes o después de las solicitudes de las pistas 124, 105, 181 y 115 disminuiría enormemente el movimiento de la cabeza y el tiempo

para servir cada solicitud.

Planificación SSTF: primero la de menor tiempo de posicionamiento

La estrategia SSTF (shortest service time first) consiste en atender la petición que requiere el menor movimiento de la cabeza de lectura/escritura desde su posición actual. De esta forma se elige la opción que incurre en el menor tiempo de búsqueda. Esto sin embargo no garantiza que el tiempo medio de búsqueda a lo largo de un número de movimientos sea mínimo. Como la cabeza se puede mover en las dos direcciones es posible emplear un algoritmo de "tirar la moneda" para resolver los casos de distancias iguales.

En el ejemplo de la cola de solicitudes anterior, la pista más próxima a la posición inicial (95) es la pista 105. Una vez situados en esta pista, la siguiente petición más próxima es la de la 115 y así sucesivamente tal como se muestra en la Tabla 6-7. Este método de planificación da como resultado una longitud media de búsqueda de 29,1, lo que representa una mejora de más del 50% respecto al caso anterior.

Próxima pista a la que se accede	105	115	124	142	181	59	36	22	5	
Número de pistas que se atraviesan	10	10	9	18	39	122	23	14	17	LMB = 29,1

Tabla 2.19: Algoritmo SSTF de planificación del disco

Un problema potencial que se puede presentar con el algoritmo SSTF es el *bloqueo indefinido* (cierre) de algunas peticiones. Conviene observar, que en un sistema real las solicitudes pueden llegar en cualquier momento. Por ejemplo, supóngase que se tiene en la cola dos peticiones, una para la pista 5 y otra para la 181. Si mientras se está atendiendo a la petición de la pista 5 llega la de otra que está próxima a ella, ésta será la siguiente en servirse, por lo que la solicitud de la 181 deberá esperar. Este argumento podría repetirse de forma indefinida con otras pistas cercanas entre si, lo que ocasionaría que la solicitud de la pista 181 espere indefinidamente.

Planificación SCAN

El algoritmo SCAN (rastreo) evita el bloqueo indefinido que se puede producir con la planificación SSTF. La estrategia es ir recorriendo todas las pistas en una dirección y satisfaciendo todas las peticiones que se encuentra en el camino, hasta que alcanza la última pista o no hay más solicitudes en esa dirección. En este punto se invierte el sentido del recorrido y la búsqueda prosigue de la misma forma. También se le conoce como *algoritmo del ascensor,* por su analogía a como se atienden las llamadas de servicio para desplazarse de un piso a otro en un edificio. En la Tabla 2.20 se ilustra la estrategia SCAN para el ejemplo que se viene tratando (se ha supuesto que el movimiento de la cabeza, desde la posición inicial, era en la dirección de las pistas decrecientes).

Próxima pista a la que se accede	59	36	22	5	105	115	124	142	181	
Número de pistas que se atraviesan	36	23	14	17	100	10	9	18	39	LMB = 29,5

Tabla 2.20: Algoritmo SCAN de planificación del disco

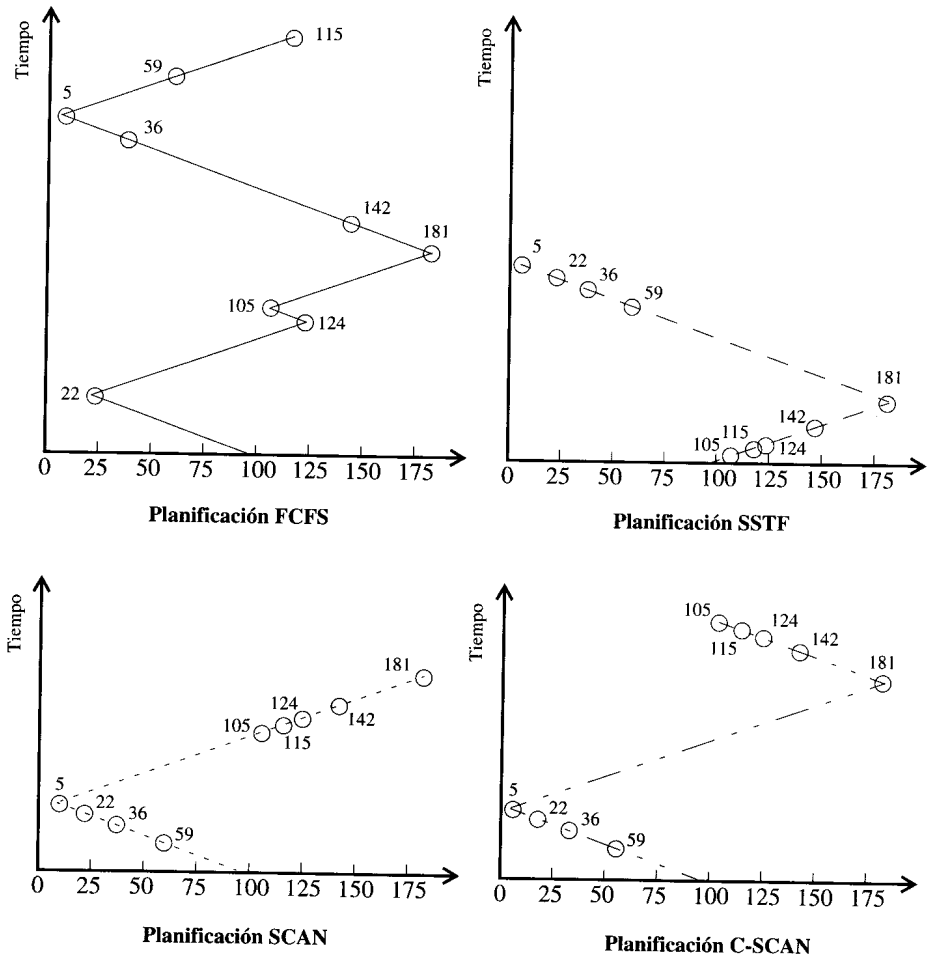


Figura 2.69: Comparación de los algoritmos de planificación del disco

Si llega una petición a la cola justo delante de la cabeza se atenderá inmediatamente, mientras que si corresponde a una posición que está por detrás deberá esperar a que se llegue a un extremo del disco y se invierta la dirección de movimiento. Esta consideración muestra que el algoritmo SCAN no explota la localidad de las peticiones al contrario que el algoritmo SSTF. No es difícil ver también que es un algoritmo de planificación, que proporciona ventajas a los procesos cuyas peticiones son a pistas que están localizadas en los cilindros más internos y externos del disco. Este problema se puede subsanar con la planificación C-SCAN.

Planificación C-SCAN

Esta estrategia restringe el rastreo a una única dirección. Así, cuando se ha visitado la última pista en una dirección, la cabeza vuelve al extremo opuesto del disco y comienza otra vez la exploración. De esta forma se

consigue reducir el retardo máximo que experimentan las nuevas peticiones. En la Tabla 2.21 se muestra la conducta del algoritmo C-SCAN para el ejemplo propuesto.

Próxima pista a la que se accede	59	36	22	5	181	142	124	115	105	
Número de pistas que se atraviesan	36	23	14	17	176	39	18	9	10	LMB = 38,0

Tabla 2.21: Algoritmo C-SCAN de planificación del disco

Básicamente, la planificación C-SCAN considera al disco como si fuera circular, con la última pista adyacente a la primera. En la Figura 2.69 se muestra una comparación de los movimientos de la cabeza de lectura/escritura para las cuatro estrategias de planificación analizadas: FCFS, SSTF, SCAN y C-SCAN.

Planificación LOOK y C-LOOK

Tal como se han presentado, las estrategias SCAN y C-SCAN mueven siempre la cabeza desde un extremo del disco al otro. En realidad ninguno de los dos algoritmos se implementan de esta manera, sino que es usual que la cabeza se mueva hasta la última petición en cada dirección. Si no hay peticiones pendientes en la dirección actual, se cambia el sentido del movimiento. A estas variantes de SCAN y C-SCAN se las denomina planificación LOOK (mirar) y C-LOOK, puesto que miran hacia adelante para ver si existe o no una solicitud antes de moverse en esa dirección.

Sección 2-9

Conclusiones

La memoria es el elemento del computador encargado del almacenamiento y subsiguiente recuperación de datos e instrucciones. La unidad de memoria de un computador presenta un espectro muy amplio de tipos, tecnologías, organización y coste. Un computador típico está equipado con una jerarquía de subsistemas de memoria, unos internos (directamente accesibles por la CPU) y otros externos (accesibles a la CPU a través de módulos de E/S). En el tema 2 se han analizado, en primer lugar, los conceptos fundamentales y las características de los diferentes tipos de memoria. El interés del tema se ha centrado en estudiar la utilización de la memoria como elemento de un computador. De toda la jerarquía de memorias, el tema se ha dedicado fundamentalmente a la *memoria interna* del computador y en concreto se estudian los siguientes tipos:

1) Memorias de semiconductor
2) Memorias caché
3) Memorias asociativas
4) Memorias compartidas
5) Memorias tipo pila

Por su relevancia práctica las memorias tipo semiconductor han sido tratadas con mayor amplitud. Haciendo un énfasis especial en su organización interna y en el diseño de módulos de memoria de mayor capacidad y longitud de palabra a partir de un módulo básico.

Las memorias caché tienen como principal objetivo acelerar la ejecución de los programas y se sitúan entre la CPU y la memoria principal del computador. Se han analizado los diferentes modos de organización de una memoria caché, las políticas de reemplazamiento de bloques, las estrategias de escritura para mantener la consistencia de la información que almacena y un análisis elemental de su rendimiento.

Las memorias asociativas son memorias direccionables por su contenido y se utilizan cuando se desea recuperar, con gran velocidad, información asociada a determinadas claves. La operación básica de recuperación de información en una memoria asociativa consiste en seleccionar las palabras de memoria que contienen una cierta información denominada clave y extraer la información asociada a dicha clave en las palabras seleccionadas. Se ha analizado brevemente el concepto de memoria compartida, que se emplean cuando más de un recurso de cálculo (por ejemplo 2 CPU's) intentan acceder a la misma unidad de memoria. Se han estudiado las memorias tipo pila que se caracterizan porque devuelven la información en sentido inverso al de su almacenamiento.

Finalmente como sistema de memoria externa se ha analizado su dispositivo más importante: el disco magnético que es la base de este tipo de memorias en casi todos los computadores.

Sección 2-10

Problemas

1) Representar un sistema digital que está constituido por una CPU, 4 memorias ROM de $1K \times 8$ bits y un bus que contiene 12 líneas de dirección y 8 líneas de datos. Añadir la lógica necesaria que seleccione uno de los 4 módulos ROM para cada una de las $4K$ direcciones.

2) Se dispone de un circuito integrado de memoria RAM de 128×8 bits.

   a) ¿Cuántos módulos se necesitan para proporcionar una capacidad de memoria de 4.096 bytes?

   b) ¿Cuántas líneas del bus de dirección se deben utilizar para acceder a los 4.096 bytes de memoria?. ¿Cuántas de estas líneas son comunes a todos los módulos?

   c) ¿Cuántas líneas deben decodificarse para seleccionar cada módulo? Especifíquese el tamaño de los decodificadores.

3) La M_p de un computador utiliza un circuito integrado de memoria RAM de $256K \times 1$ bits de capacidad.

   a) ¿Cuántos módulos se necesitan, y como se deben conectar sus líneas de dirección para proporcionar una capacidad de memoria de $256K \times 8$ bits?

   b) Lo mismo que en el apartado a) para un capacidad de memoria de $1M \times 8$ bits.

   c) Lo mismo que en el apartado a) para un capacidad de memoria de $4M \times 16$ bits.

4) Se dispone de circuitos integrados de memoria RAM de 1024×4bits. Estos chips disponen de dos señales de control, *CS* (selección del chip) y $R/\overline{W}$ (lectura o escritura). Diseñar con ellos un sistema de memoria de $2K \times 8$ bits. La memoria debe responder sólo a las direcciones 0400H a 0BFFH y se comunica con el procesador central mediante las líneas de direcciones A_{15}-A_0, de datos D_7-D_0, y las líneas de control $\overline{MEMR}$ y $\overline{MEMW}$. Para decodificar las direcciones utilizar un PLA indicando los términos producto y términos suma que debe incorporar.

5) Un pequeño computador con una longitud de palabra de 8 bits, tiene un bus de dirección de 12 bits. Las primeras 11 líneas de la dirección se utilizan para acceder a un banco de memoria de $2K$ bytes. El bit más significativo de la dirección se emplea para seleccionar un registro que recibe el contenido del bus de datos. Explíquese como se puede usar esta configuración para extender la capacidad de memoria del sistema a un total de $16K$ bytes.

6) Un computador de 8 bits de longitud de palabra, necesita 512 bytes de memoria RAM y 512 bytes de memoria ROM. Los circuitos integrados de tipo RAM y ROM que se utilizan se muestran en la Figura 2.70 (las entradas de Selección 2 que están complementadas se activan con un 0).

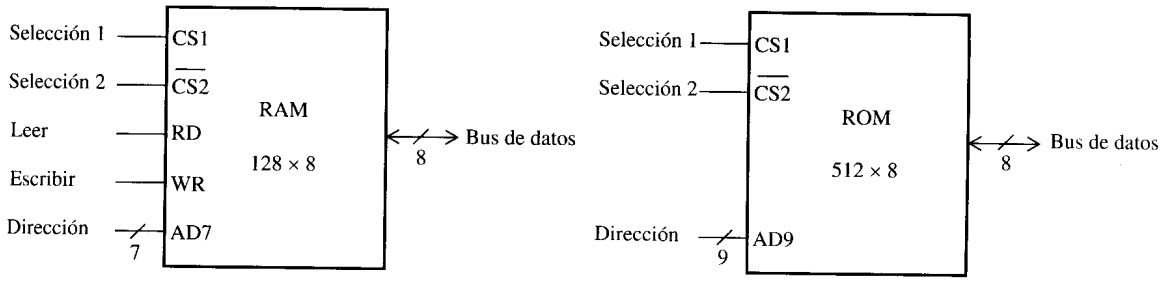


Figura 2.70: Diagrama de bloque de los módulos RAM y ROM

En la Figura 2.22 se da el mapa de direcciones de memoria que se desea. Efectuar el diseño y mostrar la conexión con la CPU.

Componente	Dirección hexadecimal
RAM 1	0000 - 007F
RAM 2	0080 - 00FF
RAM 3	0100 - 017F
RAM 4	0180 - 01FF
ROM	0200 - 03FF

Tabla 2.22: Mapa de direcciones de memoria

7) Una memoria caché con una organización asociativa por conjuntos consta de 64 bloques divididos en 4 bloques/conjunto. La memoria principal contiene $4K$ bloques de 128 palabras/bloque. Definir el formato de dirección de la memoria principal.

8) Un computador tiene una memoria principal de $32K$ palabras de 16 bits. También tiene una memoria caché de $4K$ palabras dividida en 4 bloques/conjunto con 64 palabras/bloque. Se supone que inicialmente la memoria caché está vacía. La CPU lee de forma consecutiva el contenido de las posiciones 0,1,2, ..., 4.351. A continuación repite 9 veces más esta secuencia de lectura. La memoria caché es 10 veces más rápida que la memoria principal. Estimar la mejora que se obtiene por la utilización de la memoria caché. Para el reemplazamiento de bloques en la memoria caché se emplea una estrategia LRU.

9) Sea una jerarquía de memoria de dos niveles M_c (caché) y M_p (principal) en un computador, tal como se muestra en la Figura 2.71. C_c y C_p son los costes por bit, S_c y S_p los tamaños, T_c y T_p los tiempos de acceso y h la tasa de aciertos del sistema de memoria.

   a) ¿Cuál es el coste por bit del sistema de almacenamiento?

   b) ¿Bajo qué condiciones el coste se aproxima a C_p?

c) ¿Cuál es el tiempo medio de acceso, T_a, para el sistema de memoria?

d) Sea $r = T_p/T_c$ la razón de velocidades de las dos memorias y $E = T_c/T_a$ la eficiencia de acceso del sistema. Expresar E en función de r y h, y dibujar esta relación para $r = 1$, 2 y 10.

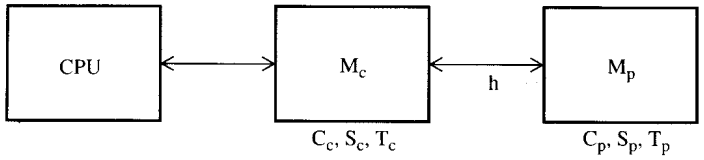


Figura 2.71: Jerarquía de memoria de un computador

10) Una memoria caché asociativa por conjuntos dispone de 2 conjuntos y utiliza bloques de 4 palabras. La memoria caché puede contener un total de 2.048 palabras de la memoria principal que tiene un tamaño de $128K \times 32$ bits.

a) Dar toda la información necesaria para construir la memoria caché.

b) ¿Cuál es el tamaño de la memoria caché?

11) Un computador tiene una unidad de memoria de $64K \times 16$ bits y una memoria caché de $1K$. La memoria caché utiliza correspondencia directa, con un tamaño de bloque de 4 palabras.

a) ¿Cuántos bits hay en los diferentes campos del formato de dirección?

b) ¿Cuántos bits hay en cada palabra de la memoria caché, y cómo se dividen según su función?

c) ¿Cuántos bloques puede contener la memoria caché?

12) Obtener la función lógica del circuito de coincidencia de una palabra, en una memoria asociativa que tiene un bit de etiqueta que indica si la palabra está activa o inactiva.

13) ¿Qué lógica adicional se necesita para dar un resultado de no coincidencia, en una palabra de una memoria asociativa, cuando todos los bits del registro de máscara M son ceros?

14) En una memoria asociativa dibujar los siguientes diagramas lógicos:

a) Todas las celdas de una palabra, incluyendo la lógica de lectura/escritura de la Figura 2.52 y la lógica de coincidencia de la Figura 2.53.

b) Todas las celdas de una columna. Incluir una línea de salida común para todos los bits que pertenecen a la misma columna.

c) A partir de los diagramas obtenidos en a) y b), mostrar que si se conecta la salida M_i a la línea de "lectura" de la misma palabra, ésta se leerá a condición de que sólo se produzca una coincidencia con el contenido del registro argumento (una vez enmascarado).

15) Obtener la estructura lógica de una celda y de una palabra completa en una memoria asociativa que tiene una salida que indica si el contenido del registro argumento sin enmascarar es mayor que (pero no igual a) la palabra en la memoria asociativa.

Tema 3

Unidad de entrada-salida

Además de la CPU y de la memoria el tercer elemento clave de un computador es la unidad de entrada/salida (E/S). La unidad de E/S proporciona un método de comunicación eficaz entre el sistema central y el usuario. Un computador basa gran parte de su utilidad en la capacidad de recibir y transmitir información desde o hacia el exterior.

Se puede ver la E/S como un conjunto de controladores de E/S; cada controlador se conecta al bus del sistema o a un conmutador centralizado y supervisa a uno o más dispositivos periféricos. Un controlador de E/S no es simplemente una conexión física entre un dispositivo externo y el bus del sistema, sino que contiene cierta "inteligencia". Es decir, dispone de la lógica necesaria para realizar una función de comunicación entre los periféricos del computador y el bus.

Los motivos por los que los periféricos no se conectan directamente al bus del sistema son los siguientes:

a) Existe una gran variedad de periféricos con diversos métodos de operación. Por este motivo, no es práctico incorporar dentro de la CPU la lógica que se necesita para controlar un conjunto tan amplio de dispositivos.

b) La velocidad de transferencia de datos de los periféricos suele ser mucho menor que la de la memoria con la CPU. Sería pues una mala estrategia utilizar el bus del sistema (de alta velocidad) para comunicarse directamente con un periférico.

c) Los periféricos a menudo utilizan formatos de datos y longitudes de palabras distintas a las del computador al que están conectados.

Estas razones justifican la existencia de los controladores de E/S, que tienen dos grandes funciones: (ver Figura 3.1)

- Comunicarse con la CPU y la memoria a través del bus del sistema.
- Comunicarse con uno o varios dispositivos periféricos mediante enlaces de datos adaptados a cada periférico.

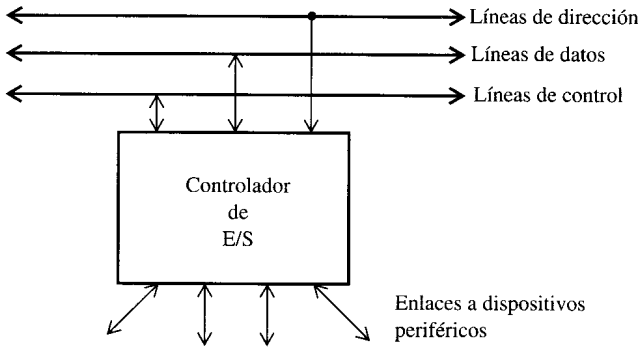


Figura 3.1: Modelo genérico de un controlador de E/S

El tema comienza con una breve introducción a los dispositivos externos mostrándose en términos muy generales la naturaleza de estos dispositivos. Su comunicación con el controlador de E/S pone en juego diferentes categorías de señales: datos, control y estado del dispositivo. Se presenta una visión unificada de la estructura y función de un controlador de E/S.

El núcleo central del tema lo constituye el estudio de los mecanismos por los que un controlador de E/S interacciona con el resto del computador. Se utilizan las siguientes técnicas:

a) *E/S controlada por programa.* La CPU ejecuta un programa que tiene el control completo y directo de la operación de E/S. Cuando la CPU transfiere una orden al controlador de E/S debe esperar hasta que acabe dicha operación.

b) *E/S por interrupciones.* La CPU envía una orden de E/S y continúa ejecutando otras instrucciones hasta que es interrumpida por el controlador de E/S cuando ha finalizado su tarea. La CPU no malgasta ahora su tiempo esperando a que se realice una operación de E/S, lo que aumenta el rendimiento global del sistema.

c) *Acceso directo a memoria* (DMA). Con esta técnica al controlador de E/S se le permite el acceso directo a memoria a través del módulo DMA. Se pueden transferir bloques de datos a o desde memoria sin intervención directa de la CPU, excepto en los instantes iniciales y finales de la transferencia.

El tema finaliza con la introducción del concepto de procesador de E/S (PE/S). El PE/S es una extensión natural del acceso directo a memoria. Un PE/S tiene la capacidad de ejecutar instrucciones de E/S lo que le da un control completo sobre dicha operación. La CPU no ejecuta ahora las instrucciones de E/S, que se almacenan en memoria principal para ejecutarse en un PE/S. La CPU comienza una transferencia de E/S al enviar una orden al PE/S para que ejecute el programa correspondiente. Se trata pues de un potente controlador de E/S capaz de ejecutar un programa.

Sección 3-1

Dispositivos externos

Un computador carecería de utilidad si no dispusiera de alguna unidad de entrada/salida. Las operaciones de E/S se pueden realizar mediante una gran variedad de dispositivos externos que proporcionan diferentes formas de intercambiar datos entre el mundo exterior y el computador. Un dispositivo externo (también llamado *periférico*) se conecta al computador a través de un enlace con un controlador de E/S (ver Figura 3.1). El enlace se utiliza para intercambiar datos e información sobre su control y su estado, entre el controlador de E/S y el dispositivo externo.

En un sentido amplio los periféricos se pueden clasificar en cuatro categorías:

1) *Adaptados al usuario*: es decir, que son apropiados para comunicar información al usuario. Ejemplos de esta clase son: las consolas, los terminales de vídeo y las impresoras.

2) *Adaptados a la máquina*: que permiten comunicarse con el sistema. A este tipo pertenecen las unidades de disco y de cinta. Conviene observar que se está tratando a estas unidades como dispositivos de E/S, mientras que en el tema 2 se consideraron como elementos de memoria. Desde un punto de vista funcional evidentemente forman parte de la jerarquía de memorias, pero estructuralmente están supervisados por un controlador de E/S.

3) *De comunicación*: que están preparados para transmitir información a dispositivos remotos. Estos dispositivos remotos pueden estar adaptados al usuario, a la máquina o incluso ser otro computador.

4) *De interacción con el medio externo*: que se utilizan en general para transmitir información a elementos actuadores y recibir información de elementos sensores en un sistema de control en el que el computador actúa como controlador.

En la Figura 3.2 se muestra en términos muy generales la naturaleza de un dispositivo externo.

La comunicación con el controlador de E/S se efectúa mediante señales de datos, de control y de estado del dispositivo. Los *datos* están constituidos por un conjunto de bits que se envían o reciben desde el controlador de E/S. Las *señales de control* determinan la función que realiza el dispositivo, tales como:

a) Enviar el dato al controlador de E/S (Leer)

b) Aceptar el dato del controlador de E/S (Escribir)

c) Realizar alguna función de control particular para ese dispositivo (por ejemplo, posicionar la cabeza de un disco)

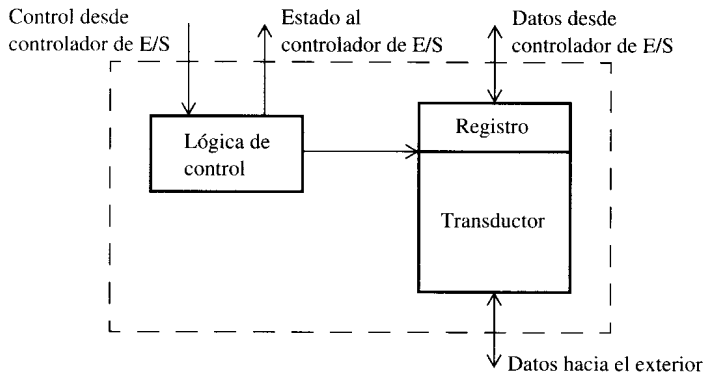


Figura 3.2: Dispositivo externo

Las *señales de estado* indican el estado en que se encuentra el dispositivo. Así la señal de *Preparado/No preparado*, informa si el dispositivo está listo para la transferencia de los datos (esta señal puede indicar por ejemplo si hay o no papel en una impresora, si un lector de CD-ROM tiene insertado un disco o si el dispositivo está ocupado en la realización de una tarea previa).

La *lógica de control* asociada con el periférico gobierna su funcionamiento en respuesta a las órdenes enviadas por el controlador de E/S. En una operación de salida los *transductores* convierten las señales eléctricas a otras formas de energía y a la inversa en una operación de entrada. Normalmente el transductor lleva asociado un registro que almacena temporalmente el dato que se transfiere entre el controlador de E/S y el mundo exterior.

Una característica que es común a todos los dispositivos periféricos es que su velocidad de trabajo es muy inferior a la que posee un computador. Esta limitación viene impuesta, en la mayoría de los casos, por la naturaleza mecánica de algunas partes de los mismos.

Sección 3-2

Controlador de E/S

Un controlador de E/S es el módulo del computador responsable del control de uno o más dispositivos externos y del intercambio de datos entre dichos periféricos con la memoria principal o con los registros de la CPU. Por esta razón, el controlador de E/S debe poseer una interfaz interna al computador (para su conexión con la CPU y con la memoria principal), y una interfaz externa al computador (para su conexión con el dispositivo externo).

3.2.1 Funciones del controlador de E/S

Las principales funciones de un controlador de E/S pertenecen a una de las categorías siguientes:

- Control y temporización
- Comunicación con la CPU
- Comunicación con el dispositivo externo
- Almacenamiento temporal de datos
- Detección de errores

En cualquier instante de tiempo la CPU se puede comunicar con uno o más dispositivos externos de manera no prefijada, dependiendo únicamente de las necesidades de E/S del programa que se esté ejecutando. Los recursos internos del computador, tales como la memoria principal y el bus del sistema, se pueden compartir por una serie de actividades entre las que se incluye la E/S de datos. Debido a esto, la función de E/S requiere un mecanismo de *control y temporización* que coordine el intercambio de información entre los recursos internos y los dispositivos externos. Por ejemplo, el control de la transferencia de datos entre un dispositivo externo y la CPU necesita la siguiente secuencia de pasos:

1) La CPU pide al controlador de E/S que compruebe el estado del dispositivo al que está conectado.
2) El controlador de E/S devuelve el estado del dispositivo.
3) Si el dispositivo está operativo y preparado para transmitir, la CPU solicita la transferencia del dato mediante una orden al controlador de E/S.
4) El controlador de E/S obtiene el dato del dispositivo externo.
5) El dato se transfiere desde el controlador de E/S a la CPU.

Si el sistema emplea un bus, cada una de las interacciones entre la CPU y el controlador de E/S implica uno o más arbitrajes para conseguir el acceso al bus. Este escenario simplificado muestra también que el

controlador de E/S debe tener la capacidad de comunicarse tanto con la CPU como con el dispositivo externo. La *comunicación con la CPU* requiere:

a) *Decodificación de la orden*: El controlador de E/S acepta órdenes de la CPU. Estas órdenes se envían generalmente por el bus de datos. Por ejemplo, un controlador de E/S de una unidad de disco podría aceptar las siguientes órdenes: "leer sector", "escribir sector", "buscar número de pista" etc. El bus de contol se emplea para gestionar la comunicación que se produce.

b) *Datos*: El intercambio de datos entre la CPU y el controlador de E/S se realiza mediante el bus de datos.

c) *Información sobre el estado*: A causa de la lentitud de los periféricos es importante conocer el estado del controlador de E/S. Por ejemplo, si se le pide al controlador de E/S que envíe un dato a la CPU (operación de lectura) puede ocurrir que no esté preparado para hacerlo porque aún se está procesando la orden previa de E/S. De esta situación se informa con una "señal indicativa del estado" del controlador de E/S. Señales usuales son "ocupado" y "preparado". Pueden haber también señales de estado que indican los diferentes tipos de errores que se presentan.

d) *Reconocimiento de la dirección*: Cada palabra de memoria tiene asociada una dirección y también la poseen cada uno de los dispositivos de E/S. El controlador de E/S reconoce una dirección única para cada uno de los periféricos que controla.

Por otra parte, el controlador de E/S realiza la *comunicación con el dispositivo*. Esta comunicación comprende órdenes, información del estado del dispositivo y datos (ver Figura 3.2).

Una tarea esencial de un controlador de E/S es el *almacenamiento temporal de los datos*. La necesidad de esta función surge por las diferentes velocidades de transferencia de datos que tienen los distintos dispositivos (ver Tabla 3.1).

Dispositivo	Transferencia de datos (bytes/seg)
Instrumentos de medida	0.1 - 1000
Pantalla (en modo texto)	10 - 2000
Impresora de línea	220 - 88000
Línea de comunicación	30 - 200K
Cinta magnética	15K - 3M
Disco magnético	30K - 10M
Memoria principal	1M - 100M

Tabla 3.1: Rango de velocidades de transferencias de datos típicas

Se ve que la velocidad de transferencia de datos de los periféricos es algunos órdenes de magnitud más lenta que en el caso de la memoria principal o por supuesto que en el de la CPU. El almacenamiento temporal en el controlador de E/S sirve para adecuar las diferencias de velocidades entre la interfaz interna con el computador (conexión a la memoria principal y a la CPU) y la interfaz externa (conexión con el dispositivo).

El controlador de E/S es a menudo responsable de la detección de errores, y de informar a la CPU cuando ocurren. Una clase de errores incluye anomalías mecánicas y eléctricas transmitidas por el propio dispositivo (por ejemplo "papel atascado", "sector del disco en mal estado" etc). Otra clase consiste en cambios en la secuencia de los bits que se transmiten desde el dispositivo al controlador de E/S. Con frecuencia se utiliza una codificación de la información que permite detectar errores simples en la transmisión. Un ejemplo es el uso de un bit de paridad que se añade a los 7 bits que ocupa el código ASCII de cada carácter. El octavo bit se pone a 0 ó a 1, para que el número total de 1's en el byte sea par (paridad par) o impar (paridad impar). Cuando se recibe un byte, el controlador de E/S comprueba la paridad para determinar si ha ocurrido un error.

3.2.2 Estructura del controlador de E/S

Los controladores de E/S varían considerablemente en complejidad y en el número de dispositivos externos que pueden controlar. En la Figura 3.3 se muestra un diagrama de bloques genérico de un controlador de E/S.

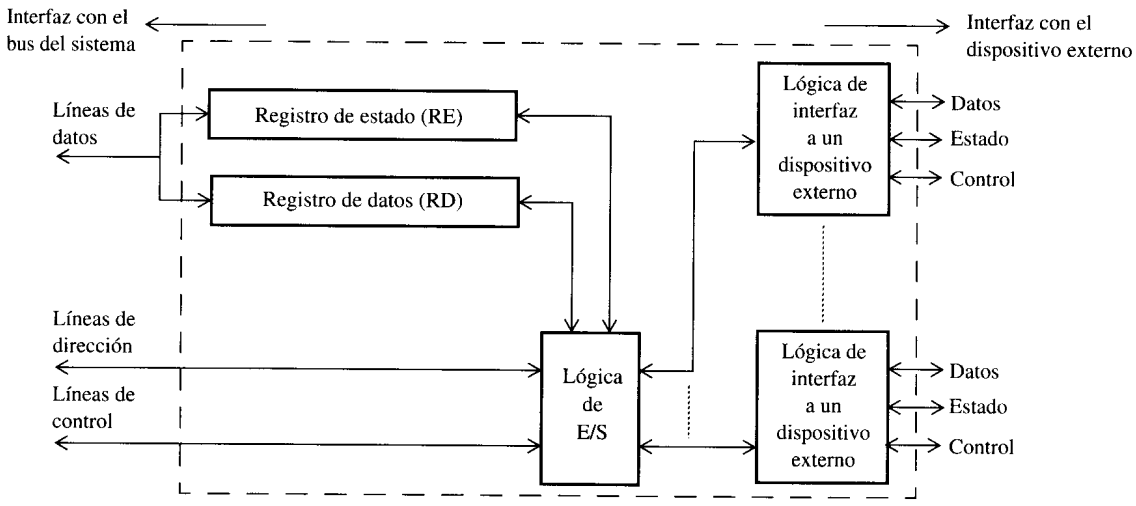


Figura 3.3: Diagrama de bloques de un controlador de E/S

El controlador de E/S se conecta con el resto del computador a través del bus del sistema. Los datos que se transfieren al controlador o desde el controlador se almacenan temporalmente en uno o más registros de datos. También puede haber más de un registro de estado que proporcione la información del estado actual. El registro de estado a veces funciona como un registro de control que acepta información proveniente de la CPU. A este conjunto de registros genéricamente se les suele llamar *puertos* del controlador. La lógica que hay dentro del módulo interacciona con la CPU mediante un conjunto de líneas de control. Estas líneas las utiliza la CPU para enviar órdenes al controlador de E/S.

Algunas de las líneas de control son utilizadas por el controlador de E/S (por ejemplo, para cuestiones de arbitraje o informar del estado en que se encuentra). El controlador debe ser capaz de reconocer y generar las direcciones asociadas con los dispositivos que controla. Cada controlador tiene asociada una única dirección, o un conjunto de ellas si controla a más de un dispositivo externo. Asimismo, el controlador de E/S contiene la lógica específica para la interfaz con cada periférico que controla.

3.2.3 Estructura del sistema de E/S

La CPU además de comunicarse con la unidad de E/S también debe hacerlo con la unidad de memoria. La forma de realizar esta comunicación, como ya se ha visto, es mediante buses que contienen líneas de dirección, de datos y de control. Hay tres maneras de utilizar el bus para interconectar la CPU con la memoria y con la unidad de E/S:

1) Utilizar dos buses independientes, uno para la memoria y otro para el sistema de E/S.
2) Utilizar un bus común para la memoria y el sistema de E/S, pero con líneas de control independientes para cada uno.
3) Utilizar un único bus con líneas de control también comunes.

En el primer método el computador tiene dos conjuntos de líneas de datos, dirección y control independientes, uno para acceder a la memoria (bus del sistema) y otro para acceder al sistema de E/S (bus de E/S). Esta estrategia se emplea en aquellos computadores que, además de la CPU, disponen de un procesador de E/S (PE/S).

La memoria se comunica tanto con la CPU como con el PE/S por el bus del sistema. El PE/S también se comunica con los dispositivos de E/S a través de un bus de E/S independiente con sus propias líneas de datos, dirección y control, al que se conectan los controladores de E/S.

Esta configuración se puede generalizar permitiendo la existencia de más de 1 PE/S (ver Figura 3.4).

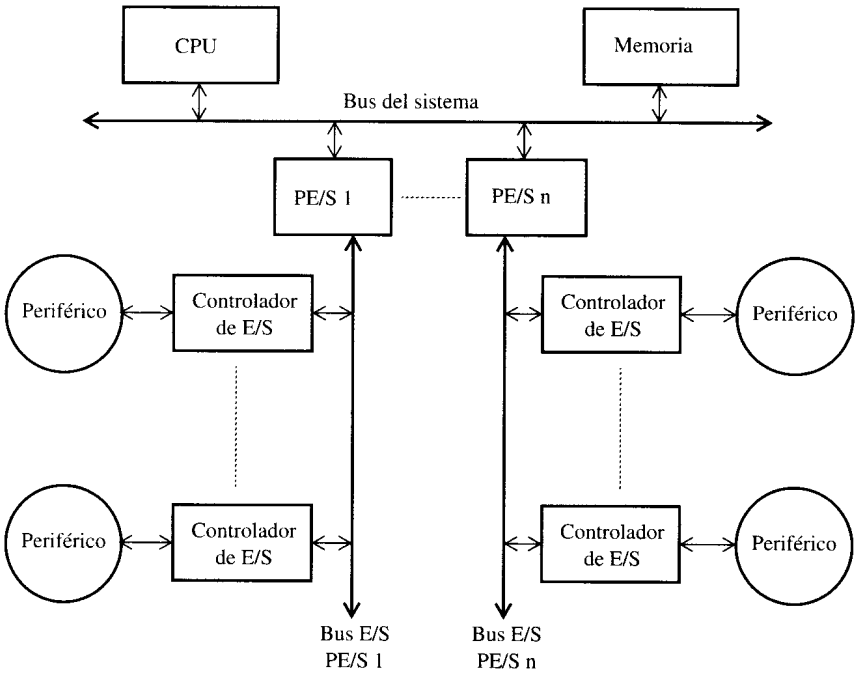


Figura 3.4: Estructura de un sistema con PE/S's

Otra denominación del PE/S es la de *canal*. En la sección 3-6 se analiza con más detalle la función de los PE/S's. Los otros dos métodos en los que la CPU, la unidad de memoria y la unidad de E/S comparten un bus común, corresponden a dos mecanismos diferentes de direccionamiento de los periféricos conocidos como :

a) E/S aislada

b) E/S localizada en memoria

Sus particularidades se verán al estudiar la E/S controlada por programa en la próxima sección.

Sección 3-3

E/S controlada por programa

Para realizar las operaciones de E/S son posibles tres técnicas. En la *E/S controlada por programa* la CPU ejecuta un programa que tiene el control directo de la operación de E/S e incluye la comprobación del estado del dispositivo, el envío de una orden de lectura o escritura y la transferencia del dato. Cuando la CPU emite una orden al controlador de E/S debe esperar hasta que finalice la operación de E/S. Si la CPU es más rápida que el controlador de E/S se malgasta el tiempo de la CPU.

En la *E/S por interrupciones* la CPU envía una orden de E/S y continúa ejecutando otras instrucciones hasta que es interrumpida por el controlador de E/S, cuando éste ha finalizado su trabajo. En estas dos técnicas, la CPU es la responsable de leer los datos de entrada de la memoria principal en una operación de salida y de escribirlos en la memoria principal en una operación de entrada. La alternativa se conoce como *acceso directo a memoria* (DMA). En este caso el controlador de E/S y la memoria intercambian datos directamente sin la intervención de la CPU (ver Tabla 3.2). En esta sección se estudia la E/S controlada por programa. Los otros dos métodos de E/S se presentan en las secciones siguientes.

	Sin interrupciones	***Con interrupciones***
Transferencia de E/S a memoria a través de la CPU	E/S controlada por programa	E/S con interrupciones
Transferencia de E/S a memoria directa		Acceso directo a memoria (DMA)

Tabla 3.2: Mecanismos básicos de E/S

Cuando se emplea la E/S controlada por programa, es el computador quien "adapta" su velocidad de trabajo a la del periférico. Si la CPU está ejecutando un programa y encuentra una instrucción de E/S, envía una orden al controlador de E/S adecuado. Este controlador realiza la acción pedida y a continuación modifica el contenido de su registro de estado RE (ver Figura 3.3) y no efectúa ninguna acción más para comunicárselo a la CPU (en particular no interrumpe a la CPU). Así pues, es responsabilidad del procesador el comprobar periódicamente el estado del controlador de E/S hasta que detecta que ha finalizado la operación.

Para explicar con cierto detalle la E/S controlada por programa, se analiza en primer lugar las órdenes de E/S enviadas por la CPU al controlador de E/S y a continuación las instrucciones de E/S ejecutadas por la CPU.

3.3.1 Órdenes de E/S

Para ejecutar una instrucción de E/S, la CPU envía una orden de E/S y una dirección que especifica el controlador y el periférico en particular. La CPU puede enviar al controlador cuatro tipos de órdenes:

a) *Órdenes de control*: Se utilizan para activar un periférico y decirle qué hacer. Por ejemplo, se le puede indicar a una unidad de cinta magnética que se rebobine o que se mueva un registro hacia adelante. Estas órdenes están adaptadas a cada tipo particular de periférico.

b) *Órdenes de comprobación*: Se utilizan para verificar diferentes condiciones de estado asociadas con un controlador de E/S y sus periféricos. Por ejemplo, la CPU necesitará conocer que el periférico está disponible para poder ser utilizado. También será preciso conocer si finalizó la operación de E/S previa y si ocurrieron errores.

c) *Órdenes de lectura*: Originan que el controlador de E/S obtenga un dato del periférico y lo coloque en un registro interno (registro de datos RD en la Figura 3.3). La CPU puede entonces conseguir el dato pidiendo al controlador de E/S que lo coloque sobre el bus de datos.

d) *Órdenes de escritura*: Realizan el proceso inverso de las órdenes de lectura y hacen que el controlador de E/S tome un dato del bus de datos y a continuación lo transmita al periférico.

La Figura 3.5 muestra la secuencia de una operación de lectura con E/S controlada por programa.

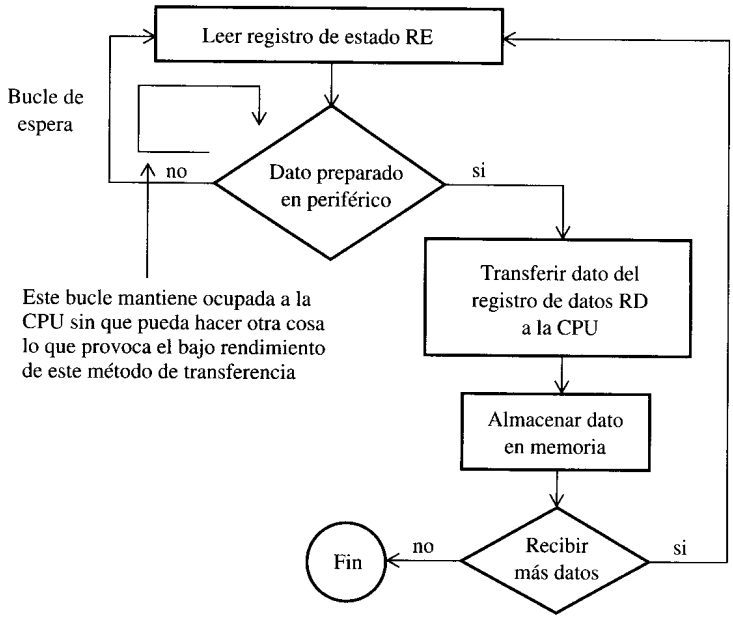


Figura 3.5: Lectura mediante E/S controlada por programa

Después de efectuar la transferencia de un dato, el computador permanece en un *bucle de espera* hasta que el periférico está preparado para realizar la siguiente transferencia. El periférico indica su disponibilidad mediante los bits de su registro de estado.

La Figura 3.5 pone de manifiesto la principal desventaja de esta técnica: *El computador no realiza ningún trabajo útil mientras permanece en el bucle de espera*, y este hecho con periféricos lentos puede repetirse miles de veces (ver Tabla 3.1).

3.3.2 Instrucciones de E/S

En la E/S controlada por programa, hay una correspondencia muy estrecha entre las instrucciones de E/S que la CPU recibe de la memoria y las órdenes de E/S que la CPU envía al controlador de E/S para su ejecución. Esto es, las instrucciones se transforman fácilmente en órdenes de E/S, y a menudo hay una relación biunívoca entre ambas. La forma de la instrucción depende de la manera en que se direccionan lo dispositivos externos.

Generalmente habrá muchos periféricos conectados al computador a través de sus controladores de E/S. Cuando la CPU emite una orden de E/S, esta orden contiene la dirección del periférico deseado y los controladores de E/S deben interpretar las líneas de dirección para determinar si la orden les compete. Cuando la CPU, la memoria principal y la unidad de E/S comparten un bus común, la forma de hacer el direccionamiento difiere según que la E/S esté localizada en memoria o esté aislada.

a) *E/S localizada en memoria*

En este caso hay un único espacio de direcciones para las posiciones de memoria y los dispositivos de E/S. La CPU trata los registros de datos y de estados de los controladores de E/S como posiciones de memoria y utiliza las mismas instrucciones máquina para acceder tanto a la memoria como a los periféricos.

Así, por ejemplo, con 12 líneas de direcciones se puede tener en total 4.096 posiciones de memoria y direcciones de E/S en cualquier combinación (el computador no distingue entre direcciones de memoria y direcciones de E/S). Sólo se necesita un conjunto de señales de lectura y escritura (ver Figura 3.6). Cada controlador de E/S se organiza como un conjunto de registros que responden a señales de lectura y escritura en el espacio normal de direcciones.

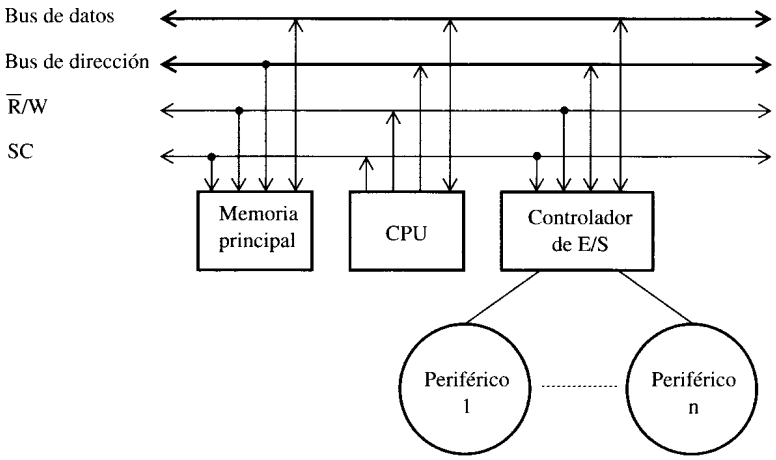


Figura 3.6: E/S localizada en memoria

Normalmente se reserva un segmento del espacio total de direcciones para los registros internos de los controladores de E/S, aunque pueden estar localizados en cualquier dirección mientras no existan palabras de memoria que respondan a la misma dirección.

b) *E/S aislada*

Cuando se tiene E/S aislada el bus del sistema dispone, además de las líneas de control de lectura y escritura en memoria, de líneas de control específicas de E/S para acceder a los periféricos. De esta forma, la línea de control especifica si la dirección se refiere a una posición de memoria o a un periférico. El rango completo de direcciones está disponible para ambos. Ahora con las 12 líneas de dirección del ejemplo anterior, la CPU puede direccionar 4096 posiciones de memoria y 4096 direcciones de E/S. Como el espacio de direcciones de E/S está aislado del de la memoria, se denomina E/S aislada (ver Figura 3.7).

En la Figura 3-8 se comparan estas dos técnicas de E/S controladas por programa. En la Figura 3.8a se muestra la E/S localizada en memoria para la lectura de un carácter que se ha pulsado en un teclado, tal como la percibe un programador. Se suponen 12 bits de dirección, con una memoria de 2048 palabras (posiciones 0 - 2047) y hasta 2048 direcciones de E/S (posiciones 2048 - 4095). Se dedican dos direcciones a la entrada del teclado.

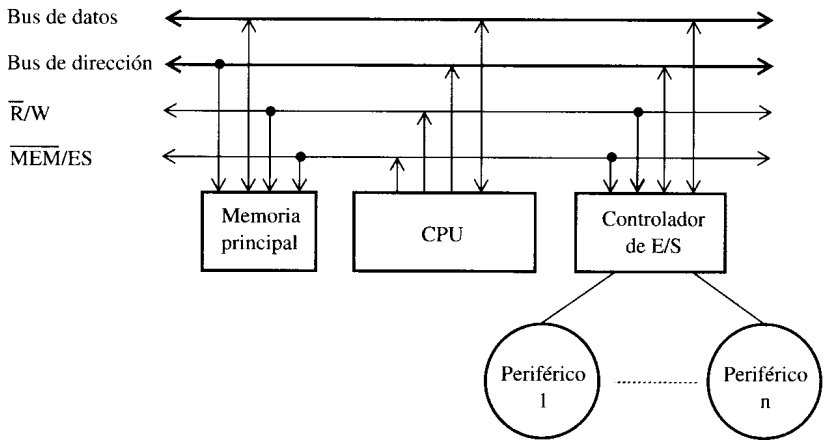
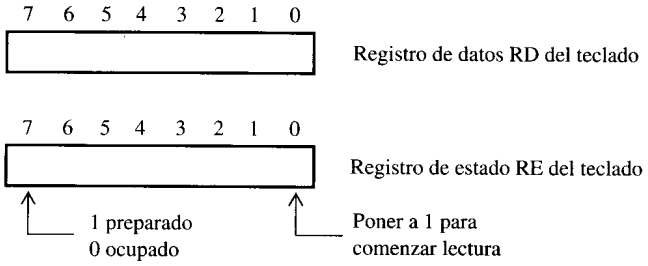


Figura 3.7: E/S aislada de memoria

La dirección 2052 se refiere al RD y la dirección 2053 al RE que también funciona como registro de control para recibir órdenes de la CPU. Se observa como la CPU permanece en el bucle de espera hasta que está disponible el byte. Con la E/S aislada (ver Figura 3.8b), los puertos de E/S son accesibles únicamente mediante órdenes especiales de E/S que activan las correspondientes líneas de control en el bus. Los computadores que poseen E/S localizada en memoria pueden usar instrucciones del tipo referencia a memoria para acceder a datos de E/S. Esto permite utilizar el mismo repertorio de instrucciones para E/S que para referencias a memoria. La ventaja es, que las instrucciones de *cargar* y *almacenar* que se emplean para leer y escribir de memoria se pueden usar también para la E/S de datos de los registros del controlador de E/S. En un computador normalmente existen muchas más instrucciones de referencia a memoria que de E/S.



a) E/S localizada en memoria

DIRECCIÓN	INSTRUCCIÓN	OPERANDO	COMENTARIO
400	Cargar AC	"1"	
	Almacenar AC	2053	Inicia lectura
402	Cargar AC	2053	Carga RE
	Bifurcar si bit 7 = 0	402	Bucle de espera
	Cargar AC	2052	Carga el dato

b) E/S aislada

DIRECCIÓN	INSTRUCCIÓN	OPERANDO	COMENTARIO
400	Iniciar E/S	6	Inicia lectura
401	Comprobar E/S	401	Comprueba fin
	Bifurcar si no preparado	6	Bucle de espera
	Leer E/S		Carga el dato

Figura 3.8: Comparación de E/S localizada en memoria y E/S aislada

3.3.3 Ejemplo: Transferencia de E/S controlada por programa en el 68000

En la Figura 3.9 se muestra un pequeño programa, hecho en el lenguaje ensamblador del 68000, que transfiere 120 caracteres desde la memoria a una impresora. El primer carácter está en la posición de memoria POSICION. El registro *A0* se utiliza como puntero a los caracteres que están almacenados en la memoria, y el registro *D0* como contador del número de caracteres ya transferidos. Los registros de datos y estado de la impresora se representan por los nombres DATO y ESTADO. La columna ciclos de memoria indica el número de referencias a memoria que se necesitan para ejecutar cada instrucción del programa. ♦

			Ciclos de memoria	
	MOVEA.L	#POSICION,A0	3	
	MOVEQ	#119, D0	1	
ESPERAR	TST.B	ESTADO	4	┐
	BPL	ESPERAR	1	│ Bucle de
	MOVE.B	(A0)+, DATO	5	│ transferencia
	DBRA	D0, ESPERAR	2	┘ de caracteres

Figura 3.9: Programa para el 68000 que imprime 120 caracteres

Sección 3-4

E/S por interrupciones

El mecanismo de E/S controlada por programa presenta una serie de inconvenientes que se pueden resumir de la forma siguiente:

1) Pérdida de tiempo en el bucle de espera.
2) Si existen programas que tienen que ejecutarse necesariamente de forma periódica, esto implica que no se puede permanecer en el bucle de espera por tiempo indefinido.
3) Hay problemas cuando se quiere atender a varios periféricos. Mientras el computador espera a que un periférico esté preparado para transmitir no puede estar atendiendo a los otros.

Un intento de solucionar estos problemas es limitar el tiempo de espera en el bucle y esto se consigue si se ejecuta sólo un determinado número de veces (ver Figura 3.10).

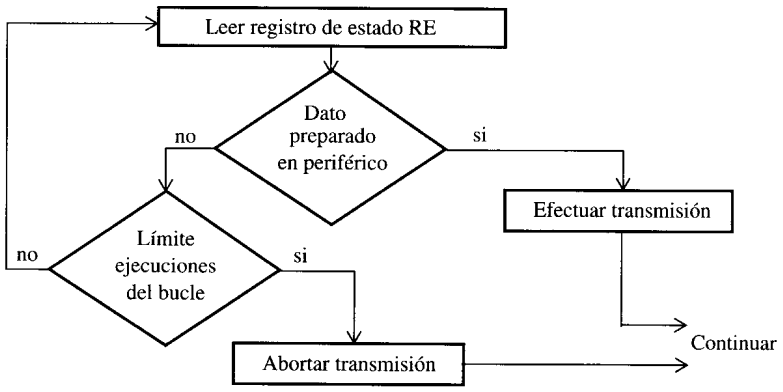


Figura 3.10: Limitación de tiempo en el bucle de espera

Esta solución sin embargo, no garantiza que se haya podido realizar la transmisión de información entre el periférico y el computador. Un planteamiento diferente, que si resuelve los problemas apuntados, lo constituye la E/S por interrupciones. *La idea básica del mecanismo de E/S por interrupciones consiste en eliminar el bucle de espera.* La CPU envía una orden de E/S al periférico y prosigue con la tarea que estaba ejecutando, en lugar de quedarse esperando a que se efectúe la operación de E/S. *Cuando el periférico está preparado para intercambiar información, fuerza una interrupción en la tarea que realiza la CPU para que atienda a la operación de E/S.* En ese momento la CPU realiza la transferencia de datos, de la misma manera que en el caso de E/S controlada por programa, y a continuación sigue ejecutando el programa que había

interrumpido (ver Figura 3.11). El periférico advierte a la CPU que está preparado para la transmisión, activando una línea especial del bus de control: La *línea de petición de interrupción PI* (ver Figura 3.12).

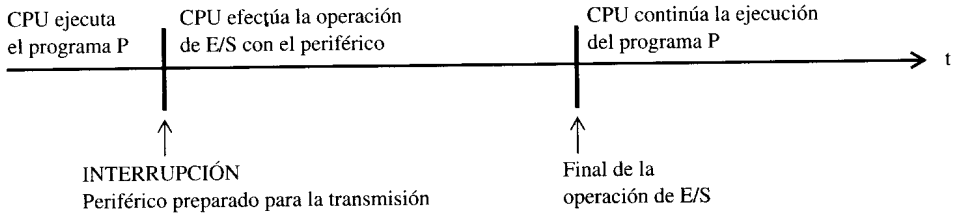


Figura 3.11: Mecanismo básico de interrupción

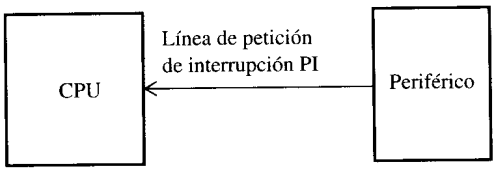


Figura 3.12: Petición de interrupción mediante una línea dedicada

En la Figura 3.13 se presenta un diagrama de flujo que ilustra la entrada de un bloque de datos mediante E/S por interrupción. Esta forma de funcionamiento tiene alguna analogía con la ejecución de un subprograma. Se "corta" la secuencia de ejecución de un programa (programa principal) para bifurcarse al subprograma desde el que posteriormente se debe retornar al punto de ruptura. No obstante, el mecanismo de interrupción presenta una diferencia sustancial, ya que el usuario no conoce a priori en que momento se va a ejecutar el programa que atiende a la interrupción. Por este motivo y para no interferir en la tarea que estaba ejecutando la CPU en el momento de ser interrumpida, es preciso guardar el contenido del PC (contador de programa) y cualquier otra información que pueda afectar a la ejecución después de retornar al programa interrumpido. En particular la CPU debería guardar el registro de estado y cualquier registro que sea accesible por programa. Esto permite que el programa original pueda continuar su ejecución sin verse afectado, salvo el retardo en el que incurre, por la aparición de la interrupción.

Conviene observar que el salvaguardar este contexto supone una sobrecarga adicional en el tratamiento de las interrupciones. En algunos sistemas se hace de forma automática por el propio sistema de interrupciones, mientras que en otros no y es el propio programa de servicio a la interrupción quien debe hacerlo. La CPU no está siempre en disposición de aceptar peticiones de interrupción por parte de los periféricos. Como ejemplos de situaciones de este tipo se pueden citar las siguientes:

a) Cuando la CPU no precisa ninguna transferencia de datos con el periférico.

b) Cuando la CPU está atendiendo la interrupción de un periférico muy prioritario, no debería aceptar peticiones de interrupción de periféricos poco prioritarios.

El usuario debe disponer de algún medio que le permita activar o desactivar las peticiones de interrupción. Una forma sencilla es utilizar instrucciones del tipo "activar interrupción" y "desactivar interrupción" que realizan esta función.

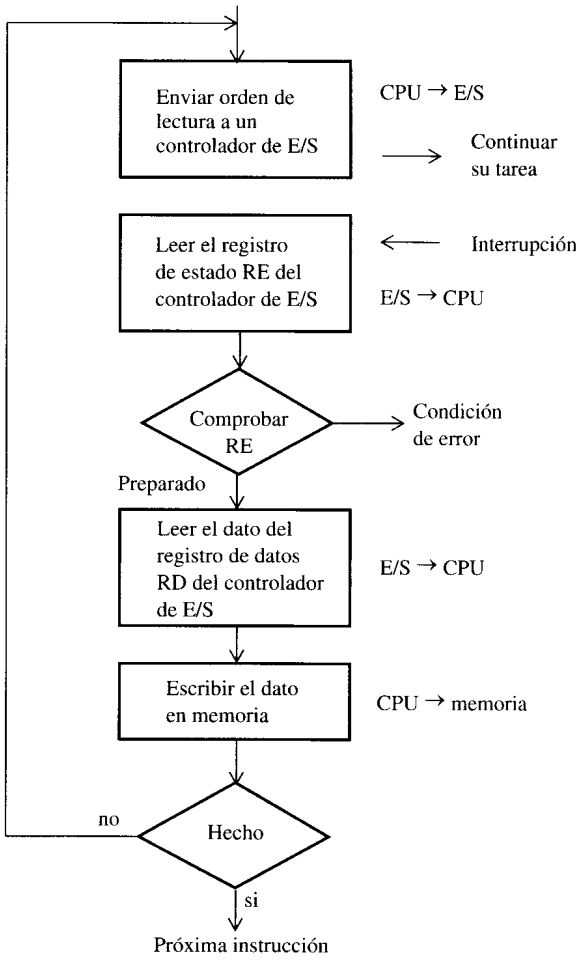


Figura 3.13: Lectura de un bloque de datos mediante E/S por interrupción

A continuación se considera con detalle el caso concreto de la petición de una única interrupción por parte de un dispositivo. Cuando un periférico activa la señal *PI*, la mantiene así hasta que sabe que la CPU ha aceptado su petición. Esto significa que *PI* estará activa durante la ejecución del programa de servicio a la interrupción, al menos hasta que se accede al periférico en cuestión. Es esencial garantizar que en este período de tiempo la señal *PI* que está activa no origina una segunda interrupción, ya que una interpretación errónea, de una interrupción simple como si fuesen múltiples peticiones, podría dar lugar a que el sistema entrase en un bucle del que no pudiese salir. Existen básicamente dos formas de conseguir esto:

1) Una vez que la CPU atiende la petición de una interrupción (reconoce que $PI = 1$), ignora esta línea hasta que ha completado la ejecución de la primera instrucción del programa de servicio de la interrupción. La primera instrucción de este programa es precisamente "desactivar interrupción" y así se asegura que no ocurrirá ninguna interrupción más hasta que se ejecute la instrucción inversa de "activar interrupción". Esta instrucción será la última antes de la instrucción de

"retorno" desde el programa de servicio al programa interrumpido. La CPU debe garantizar también que durante la ejecución de la instrucción de retorno no se produce ninguna interrupción.

2) La CPU desactiva de forma automática el sistema de interrupciones antes de comenzar la ejecución del programa de servicio. La forma de hacerlo es poner a 1 un bit de *enmascaramiento de interrupciones* en el registro de estado de la CPU (lo que equivale a ejecutar la instrucción anterior de "desactivar interrupción") antes de guardar el contexto del programa interrumpido.

   Análogamente, a la vuelta del programa de servicio de la interrupción, la CPU también de forma automática activa nuevamente el sistema de interrupciones colocando un 0 en el bit de enmascaramiento de interrupciones.

En resumen, y antes de analizar aspectos más complejos de las interrupciones, la secuencia de pasos en el tratamiento de una petición de interrupción por parte de un único periférico son:

1) Activar el sistema de interrupciones en la CPU.
2) El periférico activa la *línea de petición de interrupción PI* ($PI = 1$) cuando está preparado para comunicarse con el computador.
3) La CPU suspende en ese momento la ejecución del programa en curso. Salva el contenido del PC y de los otros registros accesibles por programa.
4) La CPU inhibe las interrupciones (bit de máscara) y comienza a ejecutar el programa correspondiente de servicio de la interrupción.
5) Se informa al periférico de que se ha reconocido su petición de interrupción (mediante una *línea de reconocimiento de interrupción*, $RI = 1$). El dispositivo desactiva *PI*, ($PI = 0$).
6) Una vez finalizado el programa de servicio de la interrupción, se activa de nuevo el sistema de interrupciones que se había inhibido en el paso 4.
7) La CPU continúa la ejecución del programa interrumpido en el punto en el que lo dejó.

```
*         Coloca el puntero en la siguiente entrada que está libre en la tabla (buffer)
*         Lee un byte de la tabla y lo transmite a la interfaz
*         Mueve el puntero a la siguiente entrada en la tabla y lo guarda en memoria
*         Retorna de la interrupción
*
*
          ORG       $001000     Origen de los datos
BUFFER    DS.B      1024        Reserva 1024 bytes para la tabla
PUNTERO   DS.L      1           Reserva una palabra larga para el puntero
SALIDA    EQU       $008000     Dirección del puerto de salida localizado en memoria
          ORG       $002000     Dirección de comienzo del fragmento de programa
*
          MOVE.L    PUNTERO, A0     [A0] ← [MS(PUNTERO)]
          MOVE.B    (A0)+, D0       [D0] ← [MS([A0])]; [A0] ← [A0] + 1
          MOVE.B    D0, SALIDA      [MS(SALIDA)] ← [D0]
          MOVE.L    A0, PUNTERO     [MS(PUNTERO)] ← [A0]
          RTE                       Retorno de la interrupción
```

Figura 3.14: Un sencillo programa para el 68000 que maneja interrupciones

La E/S por interrupciones requiere un programa más complejo que la E/S controlada por programa porque la transferencia de información tiene lugar no cuando el programador la necesita o espera sino cuando el dato está disponible. El software que se requiere para implementar una operación de E/S por interrupciones es frecuentemente parte de un sistema operativo complejo. En la Figura 3.14 se muestra un fragmento de un hipotético programa de salida de datos mediante interrupciones en el lenguaje ensamblador de la familia del 68000. Cada vez que se llama a la rutina de manejo de interrupciones, el dato se obtiene de un buffer y se pasa al puerto de salida localizado en memoria en la dirección $008000. En un sistema práctico sería necesario incluir algún tipo de test para comprobar si se está al final del buffer.

3.4.1 Clasificación de las interrupciones

El planteamiento anterior es válido cuando existe un único periférico conectado a la línea de petición de interrupciones. El problema se complica cuando hay más de un dispositivo y cada uno de ellos puede solicitar de forma independiente la petición de una interrupción. Esto supone tener que dar respuesta a las siguientes cuestiones:

a) ¿En que fase de la ejecución de una instrucción deben tratarse las peticiones de interrupción que están pendientes?

b) ¿Cómo se identifica al periférico que ha interrumpido a la CPU?

c) ¿Qué se debe hacer cuando varios periféricos hacen peticiones simultáneas de interrupción?

d) ¿Se debe permitir que un periférico interrumpa a la CPU cuando está ejecutando el programa de servicio de interrupción de otro dispositivo?

Por lo que respecta a la primera cuestión, la forma habitual de efectuar la consulta sobre las interrupciones que aún no han sido atendidas es al final de cada instrucción. La CPU examina si hay alguna petición de interrupción pendiente. Si la instrucción es de larga duración, la consulta se hace en determinados puntos de la ejecución de la instrucción (por ejemplo, en la instrucción "MOVE fuente, destino, número de palabras" que corresponde a mover bytes, se podría actualizar el contador de palabras transferidas y a continuación consultar si hay alguna interrupción pendiente).

Origen de las interrupciones	a) Externa. Las provoca un periférico b) Interna. Las provoca la CPU (ej. dividir por 0)
Número de líneas de interrupción	a) 1 línea. Solo 1 línea de petición de interrupción PI b) Múltiples líneas. PI_1, PI_2, ..., PI_n
Control de la CPU sobre la interrupción	a) Enmascarables. La CPU puede desactivarlas b) No enmascarables. La CPU no puede desactivarlas
Identificación de la fuente de la interrupción	a) Múltiples líneas. PI_1, PI_2, ..., PI_n b) Encuesta. La interrupción se identifica por programa c) Vectorizadas. La interrupción identifica al periférico
Gestión de la prioridad de la interrupción	a) Por software. Un programa determina la prioridad b) Por hardware. Un circuito determina la prioridad
Niveles de interrupción	a) Nivel único. La interrupción no puede interrumpirse b) Multinivel. Anidamiento de interrupciones

Tabla 3.3: Clasificación de las interrupciones

Las interrupciones se pueden estudiar atendiendo a múltiples criterios que no son mutuamente excluyentes entre sí. En la Tabla 3.3 se presenta una clasificación que resume los diferentes tipos de interrupciones. En cada caso se da una breve explicación que permite conocer en qué consiste. En los apartados que siguen se analizan algunos de estos tipos con un mayor detenimiento y se trata de dar respuesta a las preguntas formuladas.

3.4.2 Origen de las interrupciones

Las interrupciones pueden clasificarse de acuerdo con la naturaleza de los acontecimientos o causas que las provocan. Las *interrupciones externas* son las originadas por una causa ajena a la CPU. Por ejemplo:

- Controlador de un periférico que solicita la transferencia de un carácter o que avisa de la finalización de la transferencia de un bloque en una operación de acceso directo a memoria (ver sección 3-5).
- Fallo de la tensión de alimentación del procesador. Se produce cuando un circuito externo a la CPU detecta que la tensión suministrada por la fuente de alimentación desciende por debajo de un cierto umbral. En los milisegundos siguientes la tensión puede seguir bajando, pero hasta que el procesador deje de estar operativo aún tiene tiempo suficiente para hacer muchas cosas. El programa de servicio de la interrupción puede por ejemplo almacenar en un disco el contenido de los registros del procesador y ciertas partes de la memoria principal con información relevante para poder arrancar posteriormente el sistema con facilidad.
- Señal de reloj. La *interrupción de reloj*, también está generada por circuitos externos pero es de carácter periódico. El programa de servicio puede actualizar variables que indican el tiempo que ha pasado. En los sistemas de tiempo compartido el sistema operativo emplea este tipo de interrupción para repartir equitativamente la CPU entre los usuarios.
- En los sistemas que poseen gestión de memoria, intento de acceso a una zona que no está permitida.
- En los sistemas multiprocesadores, cuando uno de los procesadores interrumpe a cualquier otro para transmitirle un mensaje o solicitarle un recurso.

Las *interrupciones internas* se generan como consecuencia de alguna circunstancia interna de la CPU, generalmente indicativa de una situación problemática. Por ejemplo:

- *Desbordamiento en la unidad aritmético-lógica.* El programa de servicio de la *interrupción por desbordamiento* puede escribir por la pantalla un mensaje de aviso al usuario.
- *Interrupción de código de operación desconocido.* Es muy interesante cuando la CPU permite ampliaciones. Por ejemplo es bastante normal que la "unidad en coma flotante" sea opcional. Las instrucciones de operaciones en coma flotante tendrán sus correspondientes códigos binarios que la CPU no comprende si no dispone de dicha unidad, en este caso la CPU genera una interrupción de manera que será el programa de servicio quien se encargue de realizar las operaciones mediante software.
- *Interrupción de rastreo.* Se origina después de la ejecución de toda instrucción cuando la CPU está en el denominado "modo de rastreo". La CPU debe tener incluido entre sus bits de estado (por ejemplo los indicadores "*C*", "*Z*", etc., forman también parte del estado) uno que pueda ser

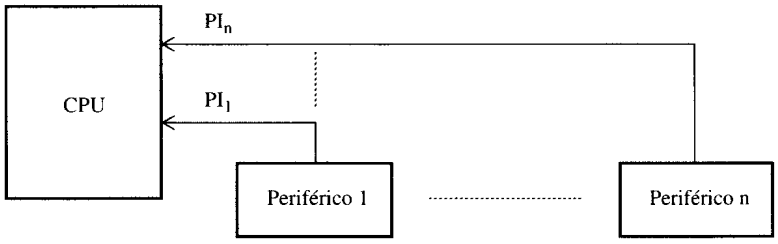
cambiado por el programador y que indique, si está a 1 que la CPU se encuentra en el modo de rastreo. En este modo siempre se genera una interrupción tras la ejecución de cada instrucción. Esto supone una ayuda a la hora de depurar los programas mediante el procedimiento de seguir paso a paso la ejecución para localizar la instrucción que provoca su funcionamiento anómalo. El programa de servicio de esta interrupción puede presentar en pantalla o enviar a la impresora un listado con todos los contenidos de los registros. El usuario mediante la instrucción que le permite poner a 0 ó a 1 el bit de rastreo puede delimitar aquellas partes del programa que desea depurar.

- *Interrupción de programa.* Son interrupciones originadas por la ejecución de instrucciones específicas situadas en el seno del programa. Su programa de servicio también debe ser escrito por el usuario y pueden realizar funciones diversas, desde simular interrupciones externas hasta simplemente sustituir la llamada a subprogramas.

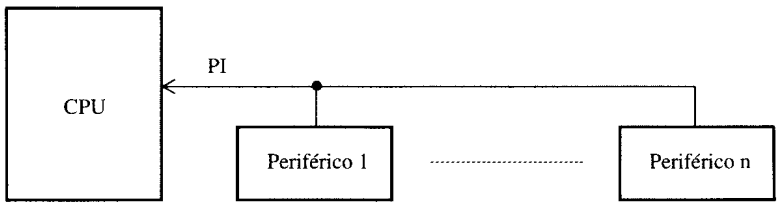
Las interrupciones externas son *asíncronas* con relación al programa, mientras que las internas son *síncronas*. Es decir, si el mismo programa se ejecuta varias veces las interrupciones internas sucederán siempre en los mismos puntos (durante la ejecución de las mismas instrucciones), mientras que los instantes de aparición de las externas pueden variar.

3.4.3 Número de líneas de interrupción

La forma más directa de identificar la fuente de una interrupción es proporcionar *múltiples líneas* de petición de interrupción PI_1, ..., PI_n entre la CPU y los controladores de E/S (ver Figura 3.15a).



a) La CPU posee varias entradas de interrupciones



b) La CPU posee una única entrada de interrupción

Figura 3.15: Identificación de la fuente de interrupción

La CPU es capaz de reconocer a través de qué entrada le ha llegado la petición de interrupción y hacer que se ejecute el correspondiente programa de servicio. Sin embargo, es poco práctico dedicar cada línea de petición de interrupción exclusivamente a un único periférico.

La situación normal es que el número de líneas de que se dispone, sea menor que el número de periféricos que pueden solicitar interrupción. Así, es preciso reunir las peticiones de interrupción de varios periféricos en una única línea (ver Figura 3.15b).

3.4.4 Control de la CPU sobre la interrupción

Existen algunos procedimientos para *activar o desactivar de forma selectiva las peticiones de interrupción* que recibe el procesador, de manera que se autoriza que determinadas solicitudes lleguen a la CPU mientras que otras quedan enmascaradas. Hay dos formas básicas de realizar este enmascaramiento:

1) *Enmascaramiento individual*

2) *Enmascaramiento por nivel*

En el primer caso, de enmascaramiento individual, cada una de las n entradas PI_i de petición de interrupción está controlada por una puerta AND de dos entradas, cuya segunda entrada es el bit correspondiente de un *registro de máscara de interrupciones* (ver Figura 3.16). Cada uno de los bits M_i del registro de máscara son accesibles por programa y en cada momento la CPU escribe en este registro el valor deseado. Las entradas de interrupción que coinciden con 1 en el registro de máscara están autorizadas, mientras que a las que le corresponden un 0 están denegadas. El registro de estado de interrupción almacena las peticiones de interrupción que han sido autorizadas por el registro de máscara. Este registro puede ser leído por la CPU con el fin de identificar el origen de la interrupción. Las peticiones de interrupción se deben resolver por programa.

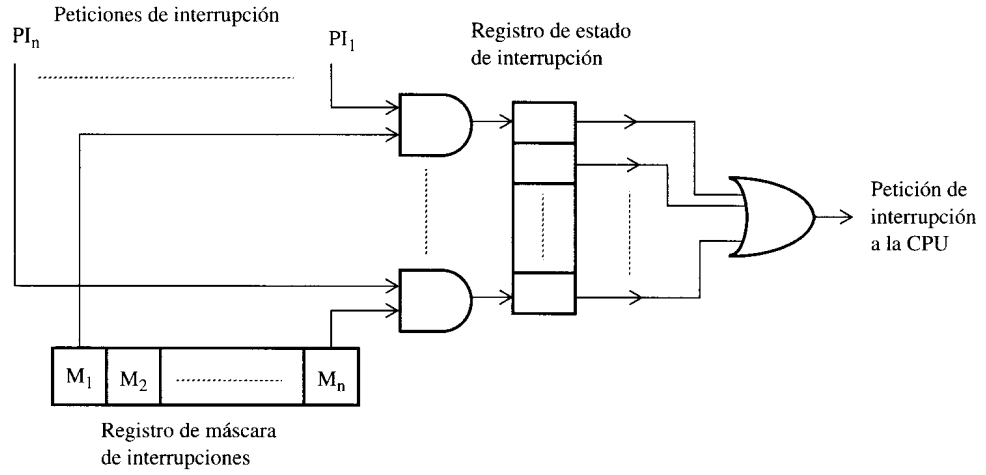


Figura 3.16: Enmascaramiento individual de cada entrada de interrupción

En el segundo caso, de enmascaramiento por nivel, las peticiones de interrupción se ordenan atendiendo a un criterio de prioridad. La CPU fija un *nivel* que puede ser modificado también por programa en cualquier momento. Todas las interrupciones que tengan un nivel superior al especificado pueden ser atendidas, mientras que las de nivel igual o inferior quedan prohibidas. En la Figura 3.17 se muestra un posible esquema que soluciona este tipo de enmascaramiento. En este ejemplo existen 4 entradas de petición de interrupción

(PI_1, PI_2, PI_3 y PI_4). La prioridad más alta corresponde a PI_1 y la más baja a PI_4. Un codificador de prioridad (que se estudiará detenidamente en el siguiente apartado) entrega a su salida el código binario de dos bits que indica la entrada de mayor prioridad que ha solicitado una interrupción. El código C que genera el codificador de prioridad se compara con otro valor A que previamente ha escrito el programa en un puerto de salida. Sólo se transmitirá la interrupción ($PI = 1$) cuando $C > A$, es decir, si el nivel de la solicitud supera al nivel fijado por el programa. Este último nivel puede ser cambiado de forma dinámica por la CPU.

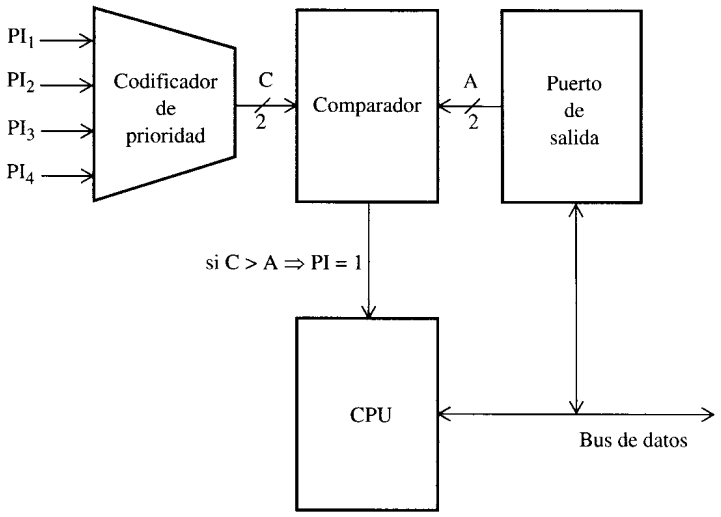


Figura 3.17: Enmascaramiento de interrupciones por nivel

3.4.5 Identificación de la fuente de la interrupción y gestión de su prioridad

Cuando en una misma línea de petición de interrupción se tienen varios dispositivos se plantea a la CPU el problema de diferenciar cuál de los periféricos conectados es el que la ha solicitado, para poder atenderle de forma específica. Hay diferentes estrategias para identificar la fuente de la interrupción y gestionar su prioridad que van desde una solución por software hasta utilizar circuitos hardware especializados en este objetivo.

Identificación de la fuente de la interrupción mediante encuesta

Una alternativa es la denominada *identificación por encuesta*. Al detectar la CPU que se ha activado la línea de petición de interrupción (es decir, $PI = 1$) lo primero que hace es ir a un programa de servicio de las interrupciones, donde interroga a cada uno de los controladores de E/S para determinar cuál de ellos originó la interrupción. La encuesta se puede realizar mediante una línea de control independiente de PI (por ejemplo "comprobar E/S"). En este caso, la CPU activa esta línea (comprobar E/S = 1) y coloca secuencialmente sobre el bus de direcciones, la dirección de cada uno de los controladores de E/S que tiene conectados, hasta que uno de ellos le responde de manera positiva (activando una determinada línea de control) si fue el que originó la interrupción. Otra forma de realizar la encuesta es que el programa de servicio examina secuencialmente (lee) el registro de estado *RE* de cada controlador de E/S hasta encontrar al que activó la línea de petición de interrupción. Una vez se ha identificado al periférico causante de la interrupción, la CPU comienza a ejecutar un programa de servicio específico para esa interrupción (ver Figura 3.18).

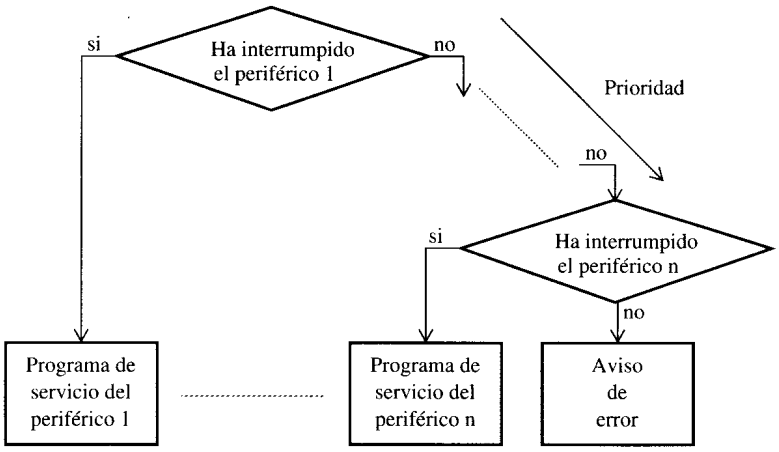


Figura 3.18: Identificación de la fuente de interrupción mediante encuesta

La desventaja que presenta el método de identificación por encuesta es el tiempo que se pierde (por programa) averiguando el periférico que ha interrumpido a la CPU. El método sin embargo garantiza un mecanismo de gestión de prioridades cuando dos o más periféricos solicitan simultáneamente una petición de interrupción. El orden en que se comprueba si el periférico ha interrumpido o no, determina la prioridad de cada interrupción. Así en la Figura 3.18 el periférico 1 tiene la máxima prioridad y el periférico n la mínima. El sistema atenderá siempre al periférico con mayor prioridad entre los que han solicitado una interrupción. Su ventaja es que se puede modificar por software la prioridad simplemente cambiando el orden de consulta a los periféricos.

Identificación de la fuente de la interrupción mediante encadenamiento

El método de *encadenamiento* (daisy chain) de establecer la prioridad, que es un mecanismo de encuesta de tipo hardware, consiste en una conexión serie de todos los periféricos que solicitan una interrupción. En la Figura 3.19 se muestra el método de conexión entre estos periféricos y la CPU. Cuando hay peticiones simultáneas está implícita la prioridad de los periféricos. El periférico con la máxima prioridad se coloca en la primera posición (periférico 1) que es el que está más próximo a la CPU seguido por los periféricos que tienen menor prioridad en orden descendente hasta el de mínima prioridad que se situa en último lugar en la cadena (periférico n).

Las líneas de petición de interrupción de todos los periféricos (PI_i, i = 1, 2, ..., n) se conectan a una puerta OR para formar la línea de petición de interrupción PI a la CPU. La secuencia de acciones que tiene lugar es la siguiente:

1) El periférico i que hace la petición provoca que $PI_i = 1 \Rightarrow PI = 1$ (cuando no hay pendiente ninguna interrupción $PI = 0$).

2) Si PI = 1 la CPU reconoce la petición de interrupción y activa la señal de *reconocimiento de interrupción RI* (RI = 1).

3) La señal RI la recibe el periférico 1 en su entrada p_e (prioridad de entrada). Si el periférico 1 no

ha originado la interrupción, propaga la señal *RI* a su terminal de salida p_s poniéndola a 1 (prioridad de salida).

4) El proceso del paso 3) se va repitiendo hasta que se llega al periférico *i* que tiene pendiente la interrupción. Este periférico bloquea la señal de reconocimiento de interrupción, poniendo un 0 en su salida p_s y procede a enviar la *dirección de su interrupción* ($dint_i$) por el bus de dirección a la CPU para informarla.

5) A partir de este punto, todos los periféricos que tienen un 0 en su entrada p_e generan un 0 en su salida p_s, lo que sirve para informar al siguiente periférico en la cadena de que la señal de reconocimiento de interrupción ha sido ya bloqueada.

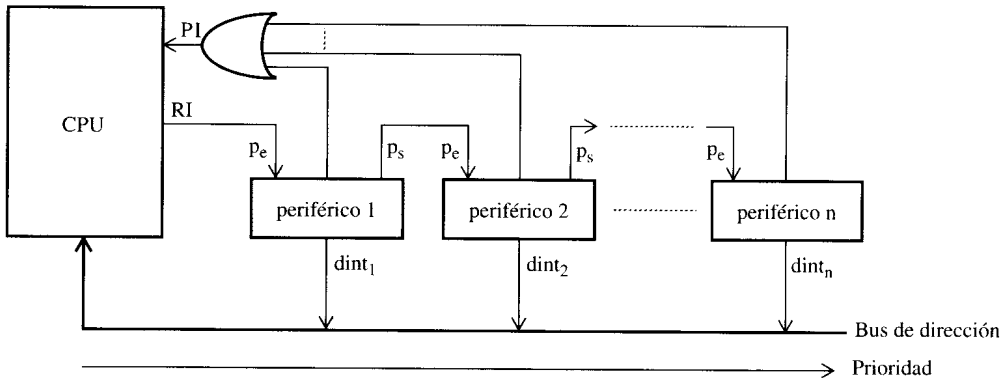


Figura 3.19: Interrupciones encadenadas

Cuando hay peticiones simultáneas está implícita la prioridad de los periféricos. El mecanismo de encadenamiento garantiza así que la máxima prioridad la posee el periférico 1 que es el que está más próximo a la CPU y la mínima el periférico *n*. En la Figura 3.20 se muestra el circuito correspondiente a la etapa iésima de la cadena.

El circuito dispone de dos entradas:

a) La señal de prioridad a la entrada del periférico p_e

b) La petición de interrupción por parte del dispositivo. El periférico pone a 1 el elemento de memoria PI_i (flip flop *RS*) cuando necesita interrumpir a la CPU

y tres salidas:

a) La señal de prioridad a la salida del periférico p_s

b) La línea de habilitación *H* de la dirección de interrupción. Cuando $H = 1$ se transfiere la dirección de la interrupción correspondiente al bus de datos

c) La señal de petición de interrupción PI_i a la CPU por parte del periférico. La salida del elemento de memoria PI_i se conecta a una puerta OR para formar la linea de petición de interrupción *PI* a la CPU que es común para todos los periféricos.

Si $p_e = 0 \Rightarrow p_s = 0$ y $H = 0$ independientemente del valor de PI_i (la señal de prioridad p_e ha sido bloqueada por un periférico con mayor prioridad). Si $p_e = 1$ y $PI_i = 0 \Rightarrow p_s = 1$ y $H = 0$ (el periférico no ha solicitado ninguna interrupción y por lo tanto propaga la señal de prioridad desde su entrada a su salida). Si $p_e = 1$ y $PI_i = 1 \Rightarrow p_s = 0$ y $H = 1$ (el periférico es el que causa la interrupción, se transfiere la dirección de la interrupción al bus de dirección y se bloquea la señal de prioridad a su salida).

El elemento de memoria PI_i se pone a 0 después de un retardo suficiente para asegurar que la CPU ha recibido la dirección de interrupción. En la Tabla 3.4 se muestra la tabla de verdad del circuito.

p_e	PI_i	p_s	H	Explicación
0	0	0	0	La señal de prioridad p_e ha sido bloqueada
0	1	0	0	ya por un periférico con mayor prioridad
1	0	1	0	Propaga la señal de prioridad $p_s = p_e = 1$
1	1	0	1	Habilita $dint_i$ (h = 1) y bloquea p_s ($p_s = 0$)

Tabla 3.4: Tabla de verdad del circuito de interrupciones encadenadas

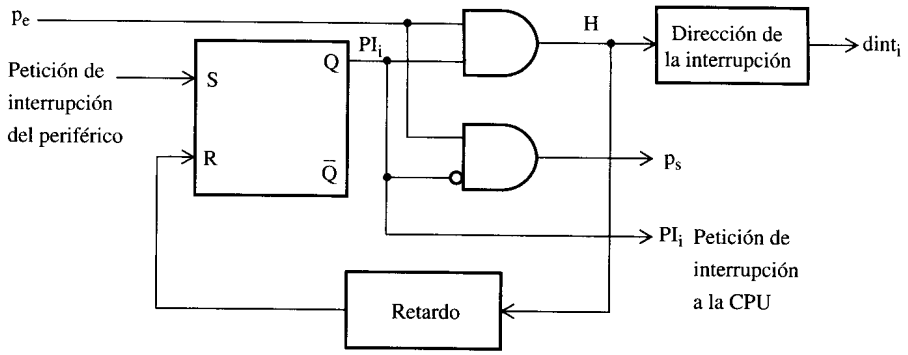


Figura 3.20: Etapa iésima de la interrupción por encadenamiento

El método de encadenamiento que se acaba de ver pertenece a la categoría de *interrupciones vectorizadas* que se caracteriza porque la petición de interrupción *PI* que se solicita a la CPU identifica al periférico que la ha realizado (a través de la dirección de interrupción). Sin embargo la estructura hardware que utiliza es de tipo serie ya que la señal de reconocimiento de la interrupción *RI* que envía la CPU va pasando de periférico a periférico hasta que localiza quien la efectuó. Es posible disponer las cosas de manera que el hardware de identificación de la interrupción lo haga de forma simultánea sobre todos los periféricos.

Identificación de la fuente de interrupción mediante hardware paralelo

En este método se utiliza un *registro de interrupción* cuyos bits se controlan de forma independiente por las señales de petición de interrupción PI_i de cada periférico. La prioridad queda establecida de acuerdo con la posición de cada bit en el registro. Además del registro de interrupción, el módulo de interrupción puede

incluir un *registro de máscara* de la misma longitud que el registro de interrupción para controlar el estado de cada petición de interrupción tal como se ha visto al tratar el enmascaramiento individual de las peticiones de interrupción. El registro de máscara se puede programar para desactivar las interrupciones de prioridad más baja cuando se está atendiendo a un periférico de mayor prioridad. También puede permitir que un periférico de mayor prioridad interrumpa a la CPU cuando se está tratando a un periférico que tiene menor prioridad.

En la Figura 3.21 se muestra la estructura lógica del circuito de prioridad paralelo, con enmascaramiento individual de las peticiones de interrupción, cuando hay 4 fuentes posibles de interrupción. De esta forma una interrupción se reconoce solamente si su bit de máscara asociado M_i se pone a 1 por el programa. El codificador de prioridad genera dos bits x e y de la dirección de interrupción que se transfiere a la CPU a través del bus de datos. El valor del resto de los bits se establecen normalmente a cero. Si hubiese más de un módulo estos bits serían distintos para cada uno de ellos. La salida *PI* del codificador se pone a 1 si ha ocurrido al menos una petición de interrupción que no está enmascarada. En la Tabla 3.5 se muestra la tabla de verdad del codificador de prioridad de 4 entradas.

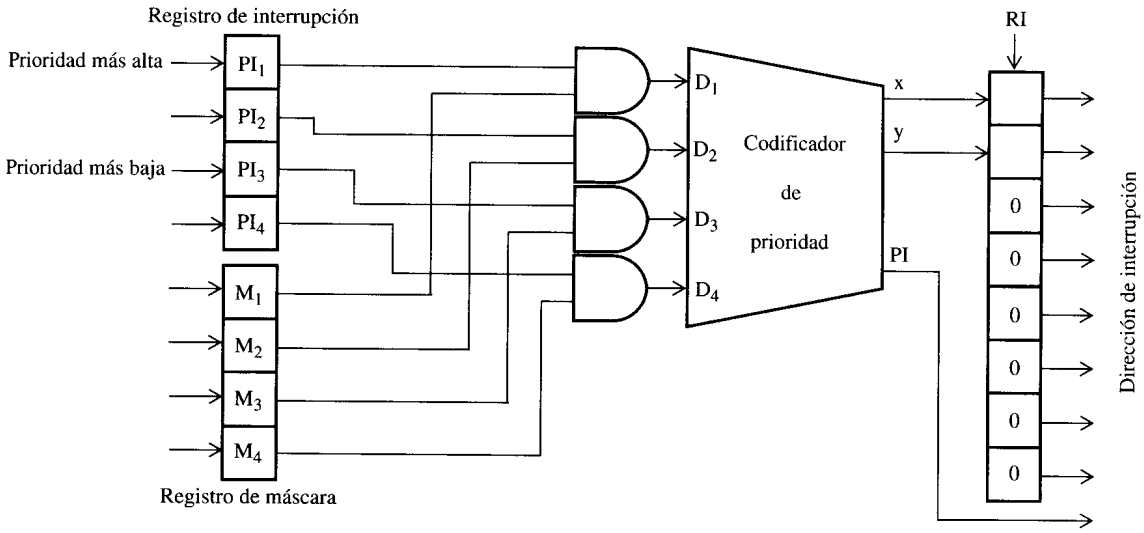


Figura 3.21: Hardware paralelo de interrupciones con prioridad

D_1	D_2	D_3	D_4	y	x	PI
1	×	×	×	1	1	1
0	1	×	×	1	0	1
0	0	1	×	0	1	1
0	0	0	1	0	0	1
0	0	0	0	×	×	0

Tabla 3.5: Tabla de verdad de un codificador de prioridad de 4 entradas

Las ×'s en la Tabla 3.5 representan que su valor puede ser indistintamente 0 ó 1. La entrada D_1 tiene la prioridad más alta, así que sin tomar en consideración las otras entradas cuando $D_1 = 1$ la salida vale $yx = 11$ (3 en binario). La siguiente prioridad la tiene D_2 (si $D_2 = 1$ y $D_1 = 0$), independientemente de los valores de las otras dos entradas de menor prioridad, en este caso $yx = 10$ (2 en binario). Este principio se aplica a las sucesivas entradas por orden de prioridad. La señal de petición de interrupción *PI* a la CPU se pone a 1 cuando una o más entradas valen 1. Si todas las entradas son *0*'s ⇒ *PI* = 0 y las otras salidas *x* e *y* del codificador no se usan (por lo que están marcadas con ×'s). Esto es así porque cuando *PI* = 0 no se transfiere la dirección de interrupción a la CPU. En la Figura 3.22 se muestran los diagramas de Karnaugh que simplifican las salidas *x* e *y*.

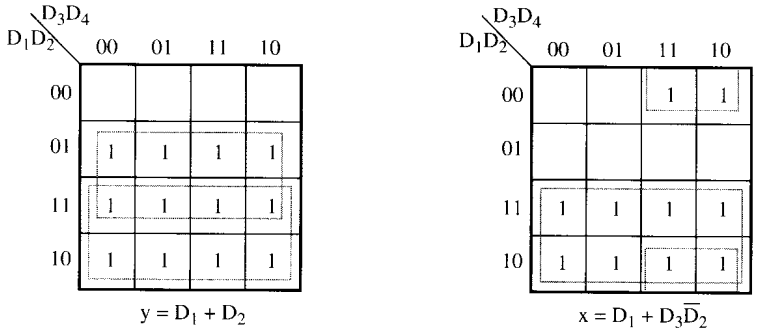


Figura 3.22: Diagrama de Karnaugh del codificador de prioridad

En la Figura 3.23 se muestra el circuito lógico que implementa al codificador de prioridad mediante las funciones booleanas siguientes:

$$x = D_1 + D_3\overline{D_2}$$
$$y = D_1 + D_2$$
$$PI = D_1 + D_2 + D_3 + D_4$$

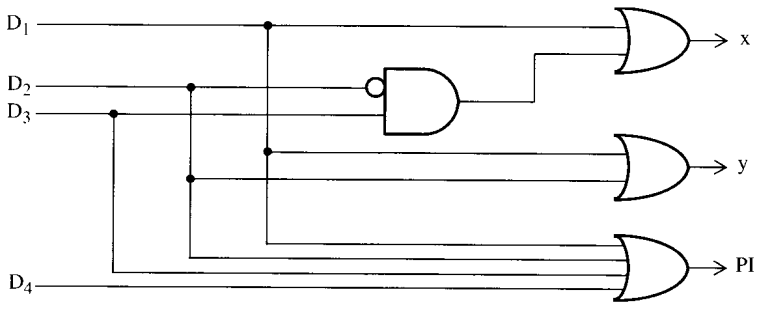


Figura 3.23: Diagrama lógico del codificador de prioridad de 4 entradas

Se observa que las dos salidas del codificador de prioridad *x* e *y* identifican cuál es la entrada con mayor prioridad que ha solicitado una interrupción y se emplean para formar la dirección de interrupción

correspondiente a cada fuente de interrupción. A los otros bits de la dirección de interrupción se les puede asignar cualquier valor. En el ejemplo que se considera, x e y pueden corresponder a los bits menos significativos y los bits más significativos de la dirección de interrupción se ponen a 0. Con esta elección, a las direcciones de interrupción de los cuatro periféricos se les asignan respectivamente los valores 0, 1, 2 y 3.

3.4.6 Niveles de interrupción

Hasta ahora se ha supuesto que el sistema de interrupciones se desactivaba durante el tiempo que se está atendiendo al programa de servicio de una interrupción. Sólo se dispone de un *único nivel* de interrupciones, de manera que una vez que se inicia la ejecución del programa de servicio de una interrupción continúa hasta su final antes de que la CPU pueda aceptar una segunda petición de interrupción.

En determinadas situaciones, cuando se dispone de periféricos muy prioritarios, esta forma de actuación supone una seria limitación y puede dar lugar a una actuación no admisible del sistema de interrupciones. Como ejemplo se puede considerar un computador que tiene un reloj de tiempo real controlado por un circuito que genera una interrupción cada microsegundo. Para que el sistema funcione correctamente es necesario asegurar que el retardo en dar respuesta a una petición de interrupción por parte del reloj de tiempo real es pequeño, en relación con el intervalo que hay entre dos peticiones sucesivas. Para conseguir este objetivo, cuando existen otros periféricos que pueden interrumpir a la CPU, es preciso que el sistema posea una estructura de interrupciones de tipo *multinivel*. Es decir, se pueden atender peticiones de interrupción del reloj de tiempo real durante la ejecución del programa de servicio de otro dispositivo.

Cuando las interrupciones pueden interrumpir a los programas de servicio de otras interrupciones se dice que se *anidan*. A cada causa de interrupción se le asigna uno de entre varios *niveles de prioridad* de modo que una interrupción sólo se atiende si su nivel es superior al de la interrupción cuyo programa de servicio se está ejecutando.

Este argumento implica que los dispositivos de E/S se organicen con una estructura de prioridades, de manera que la petición de interrupción de un dispositivo de mayor prioridad se debe aceptar cuando la CPU está atendiendo a uno de menor prioridad. Esta organización de prioridades en una estructura multinivel significa que durante la ejecución de un programa de servicio de interrupciones se podrán aceptar peticiones de interrupción de algunos dispositivos mientras que de otros no, dependiendo de sus prioridades relativas.

Para facilitar la realización de este esquema, es conveniente asignar a la CPU un nivel de prioridad que se puede modificar por programa. El nivel de prioridad de la CPU es la prioridad del programa que se está ejecutando. La CPU acepta interrupciones sólo de aquellos dispositivos que tienen una prioridad mayor que la suya. Como ejemplo de un sistema de tratamiento y servicio de interrupciones multinivel se considera el siguiente: la CPU dispone de tres líneas de interrupción PI_1, PI_2 y PI_3 de prioridad decreciente, que son enmascarables mediante 3 bits del registro de estado *RE* de la CPU (ver Figura 3.24).

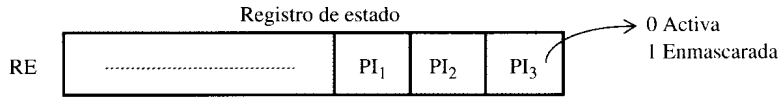
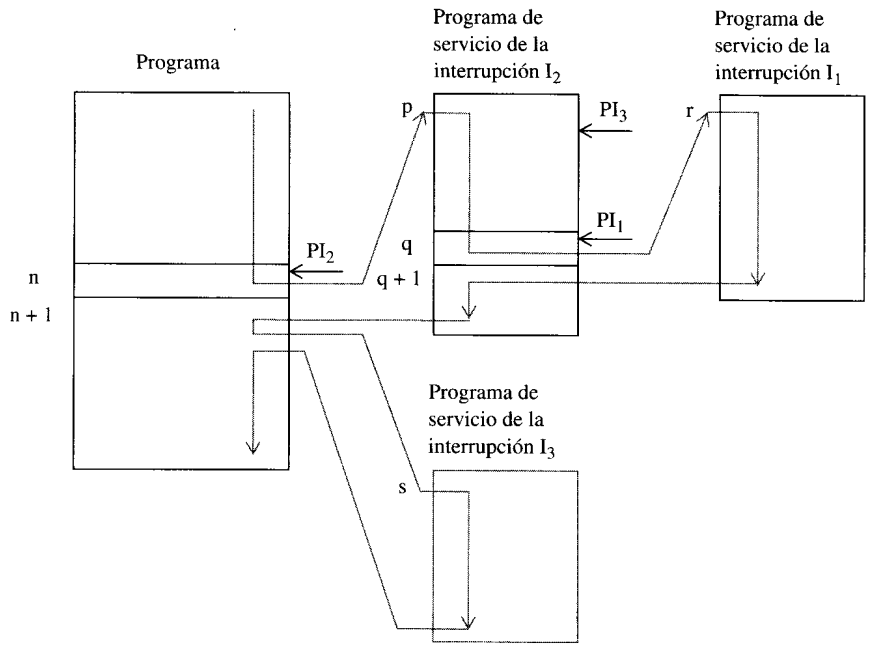


Figura 3.24: Registro de estado con múltiples líneas de interrupción

Una petición de interrupción enmascara a las demás líneas menos prioritarias. Se supone que se producen tres peticiones de interrupción en este orden: PI_2, PI_3 y PI_1 (ver Figura 3.25).



Evolución de la pila y de los registros de la CPU

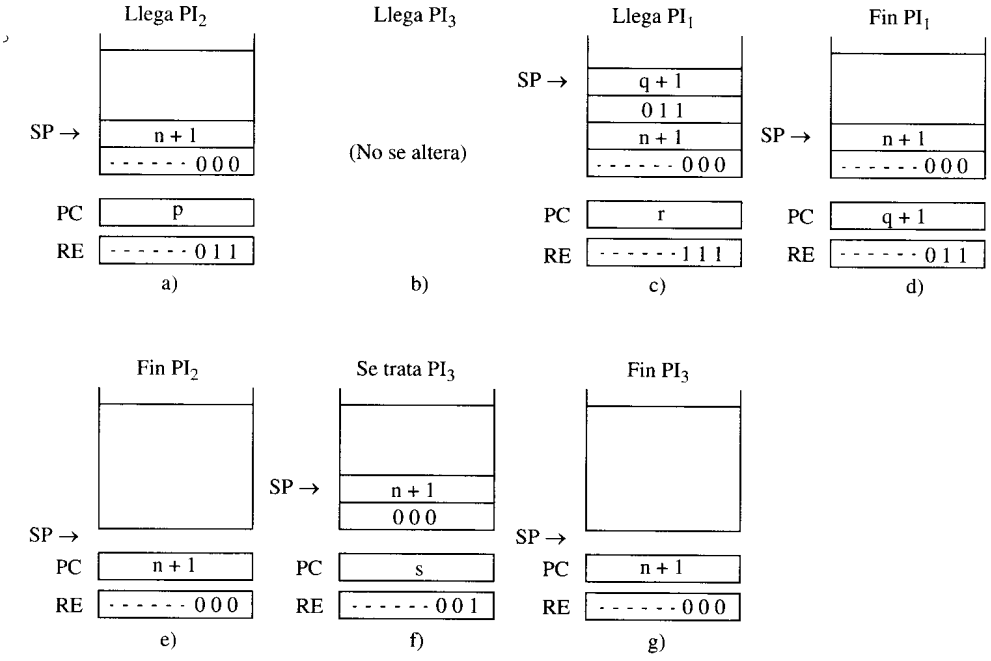


Figura 3.25: Ejemplo de tratamiento y servicio de interrupciones anidadas

Se supone que las peticiones de interrupción PI_3 y PI_1 se producen cuando se está ejecutando el programa de servicio de PI_2. En la Figura 3.25 se muestra un esquema de cómo se lleva a cabo el anidamiento de interrupciones y las modificaciones de la pila y del contenido de los registros de la CPU en esta situación en particular. De forma detallada las acciones que tienen lugar son las siguientes:

1) *Llega* PI_2. Cuando llega la petición de interrupción PI_2 la CPU se encuentra ejecutando la instrucción n del programa principal. Una vez terminada la ejecución de esta instrucción se atiende la petición PI_2 por lo que se almacena en la pila todo el contexto del programa principal que queda interrumpido. En particular se guardan el valor de RE = ...000 (los tres ultimos bits de acuerdo con el formato de la Figura 3.24 indican que las tres lineas de petición de interrupción se encuentran activadas) y el punto de continuación donde tiene que proseguir la ejecución del programa principal que es la dirección $n+1$. Para poderse transferir al programa de servicio de la interrupción I_2 se carga en el contador de programa PC su dirección de comienzo que es p. También se modifica el contenido de RE que pasa a ser, RE = ...011 (ver Figura 3.25a). Este valor de RE permite que el programa de servicio de la interrupción I_2 sólo pueda ser a su vez interrumpido por una interrupción con mayor prioridad que en este ejemplo sería I_1.

2) *Llega* PI_3. Cuando llega la petición de interrupción PI_3 la CPU se encuentra ejecutando el programa de servicio de la interrupción I_2. Al tener I_3 una prioridad menor que I_2 no se atiende en ese momento la petición y queda a la espera de poderse hacer posteriormente cuando RE la active nuevamente poniendo a 1 el correspondiente bit. Por lo tanto no se alteran el contenido de la pila ni el valor de los registros PC y RE y se sigue ejecutando el programa de servicio de la interrupción I_2 (ver Figura 3.25b).

3) *Llega* PI_1. Cuando llega la petición de interrupción PI_1 la CPU se encuentra ejecutando la instrucción q del programa de servicio de la interrupción I_2. Una vez terminada la ejecución de esa instrucción, por ser más prioritaria I_1 que I_2, se atiende de forma inmediata la petición PI_1 y se almacena en la pila todo el contexto del programa de servicio de la interrupción I_2 que queda interrumpido. En particular se guardan el valor de RE = ...011 y el punto de continuación donde tiene que proseguir su ejecución dicho programa de servicio que es la dirección $q+1$. Para poderse transferir al programa de servicio de la interrupción I_1 se carga en el contador de programa PC su dirección de comienzo que es r y se modifica el contenido de RE que pasa a ser, RE = ...111 (ver Figura 3.25c).

4) *Finaliza* PI_1. El programa de servicio de I_1 finaliza su ejecución sin que pueda ser interrumpido ya que es el de mayor prioridad. En ese momento se restaura en los registros PC y RE los valores que están almacenados en la cumbre de la pila. En PC se almacena $q+1$ que es el punto de continuación del programa de servicio de I_2, que había quedado interrumpido, y el contenido de RE pasa a ser otra vez RE = ...011 (ver Figura 3.25d).

5) *Finaliza* PI_2. El programa de servicio de I_2 finaliza su ejecución. En ese momento se restaura en los registros PC y RE los valores que están almacenados en la cumbre de la pila. En PC se almacena $n+1$ que es el punto de continuación del programa principal, que había quedado interrumpido, y el contenido de RE = ...000 (ver Figura 3.25e).

6) *Se trata* PI_3. En este momento es cuando es posible atender la petición de interrupción PI_3 que estaba pendiente al haberse puesto a 0 el bit correspondiente de *RE*. Por ese motivo se guardan en la pila el valor de *RE* = ...000 y el punto de continuación donde tiene que proseguir la ejecución del programa principal que es la dirección $n+1$. Para poderse transferir al programa de servicio de la interrupción I_3 se carga en el contador de programa *PC* su dirección de comienzo que es *s* y se modifica el contenido de *RE* que pasa a ser, *RE* = ...001 (ver Figura 3.25f).

7) *Finaliza* PI_3. El programa de servicio de I_3 finaliza su ejecución por lo que se restaura en los registros *PC* y *RE* los valores que están almacenados en la cumbre de la pila. En *PC* se almacena $n+1$ que es el punto de continuación del programa principal, que había quedado interrumpido, y *RE* = ...000 (ver Figura 3.25g). Como no hay ya pendiente ninguna interrupción el programa principal puede continuar ya su ejecución desde este punto.

3.4.7 Controlador de interrupciones

En algunas familias de procesadores, el propio fabricante proporciona un módulo denominado *controlador de interrupciones PIC* (Programmable Interrupt Controller) cuyo cometido básico es ampliar el número de líneas de interrupción de la CPU y encargarse de forma integrada de toda la gestión del sistema de interrupciones que se ha explicado en los apartados anteriores. Las funciones que realiza el *PIC* son las siguientes:

a) Identificar la fuente de la interrupción

b) Establecer las prioridades de cada periférico

c) Activar o desactivar de forma selectiva las peticiones de interrupción que recibe

d) Enviar a la CPU información sobre la petición de la interrupción y cual es el periférico que debe ser atendido

En la Figura 3.26 se muestra la arquitectura de un sistema de interrupciones que contiene un *PIC*.

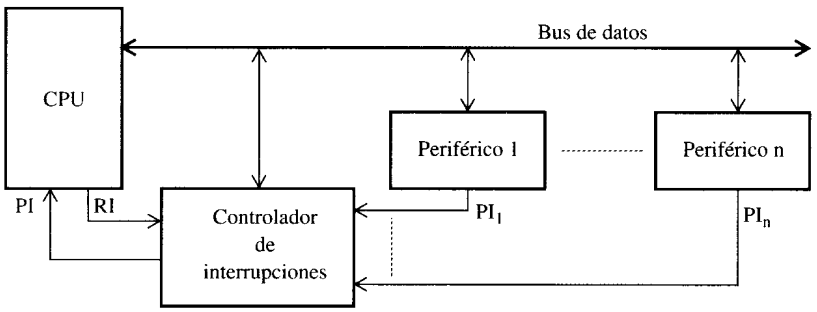


Figura 3.26: Estructura de un sistema de interrupciones con un PIC

3.4.8 Ejemplos de controladores de interrupciones

Existen en el mercado controladores de interrupciones ya encapsulados en un circuito integrado que efectúan las funciones descritas y algunas adicionales. Los fabricantes de microprocesadores suelen incluir en sus

catálogos controladores de interrupciones diseñados expresamente para funcionar de forma óptima con el sistema de interrupciones de cada microprocesador concreto. A continuación se presentan algunos ejemplos concretos de este tipo de dispositivo.

El controlador de interrupciones Intel 8259A

El procesador Intel 8086, además de un conjunto de líneas de interrupciones internas (desbordamiento en división, rebose en operaciones aritméticas, ejecución paso a paso y punto de parada) posee dos líneas *PI* de petición de interrupciones externas (INTR que es una línea de interrupción enmascarable y NMI que es una línea de interrupción no enmascarable) y una línea *RI* de reconocimiento de interrupciones (INTA). El controlador de interrupciones Intel 8259A se configura normalmente como un árbitro de interrupciones externas, que permite al procesador 8086 manejar una diversidad de dispositivos y estructuras de prioridad. Los dispositivos externos se conectan al 8259A, que a su vez lo hace al 8086. La Figura 3.27 muestra la utilización del 8259 para comunicar múltiples periféricos con el 8086.

Un módulo 8259A puede manejar hasta 8 periféricos. Si se necesitan controlar más de 8 dispositivos externos se puede emplear una configuración en cascada que permite conectar hasta 64 unidades. El 8259A se encarga sólo de la gestión de las interrupciones. Acepta las peticiones de interrupción de los periféricos que tiene conectados, determina cuál tiene la mayor prioridad y se lo indica a la CPU activando la línea *PI*. La CPU lo reconoce mediante la línea *RI*. Cuando el 8259A recibe la señal de reconocimiento de la CPU coloca la información del vector de interrupción apropiado en el bus de datos. Es entonces cuando la CPU puede procesar la interrupción comunicándose directamente con el periférico para leer o escribir el dato.

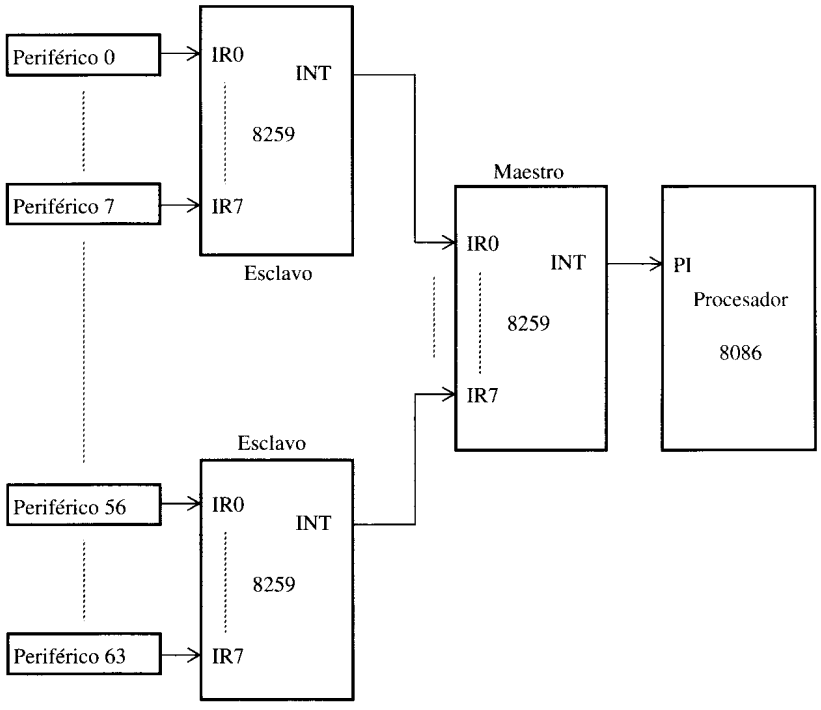


Figura 3.27: Utilización del controlador de interrupciones 8259A

El 8259A es programable. El 8086 puede modificar el contenido del registro de control del 8259A lo que determina el esquema de prioridad que se va a utilizar. Son posibles los siguientes modos de interrupción:

1) *Anidadas*. Las peticiones de interrupción se ordenan según su prioridad desde 0 (IR_0) hasta 7 (IR_7).
2) *En rotación*. En algunas aplicaciones una serie de dispositivos tienen la misma prioridad. En este modo un periférico que acaba de ser atendido recibe la prioridad más baja del conjunto.
3) *Enmascarables*. Permite a la CPU inhibir de forma selectiva las interrupciones de determinados dispositivos.

El interfaz programable Intel 8255A

Aunque no es propiamente un controlador de interrupciones el interfaz programable Intel 8255A es un ejemplo de un módulo de E/S de propósito general, que se emplea para realizar E/S controlada por programa y E/S por interrupciones conjuntamente con la popular CPU de Intel 8086. En la Figura 3.28 se muestra un diagrama de bloques del Intel 8255A.

La parte derecha del diagrama de bloques es la interfaz externa del 8255A. Las 24 líneas de E/S son programables por la CPU 8086 a través del registro de control. El 8086 puede modificar el valor de este registro para especificar una diversidad de configuraciones y de modos operativos. Las 24 líneas se dividen en tres grupos de 8 bits (*A, B, C*). Cada grupo funciona como un puerto de E/S de 8 bits. Además, el grupo *C* se subdivide en 2 grupos de 4 bits (C_A y C_B) que se pueden usar conjuntamente con los puertos de E/S *A* y *B*. Configurado de esta forma, contiene las señales de estado y de control.

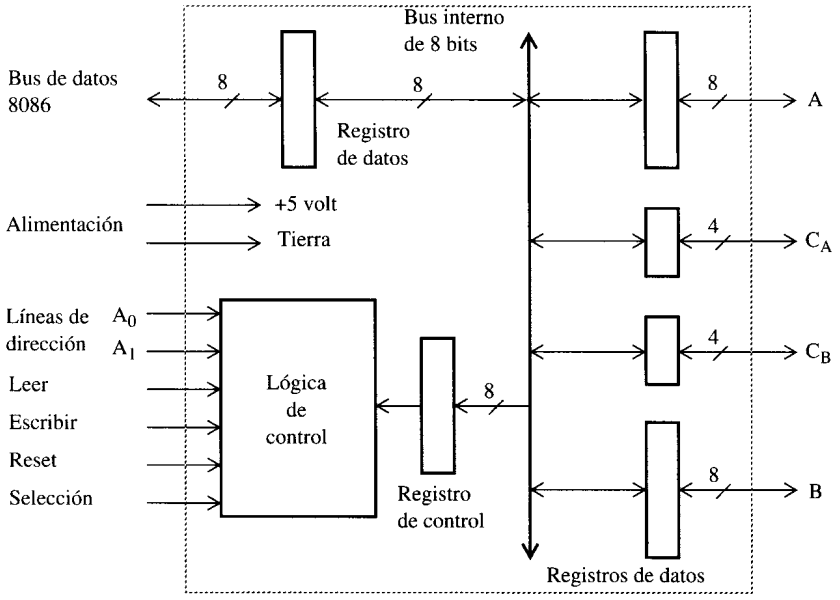


Figura 3.28: Diagrama de bloque del interfaz programable Intel 8255A

La parte izquierda del diagrama de bloques es la interfaz interna al bus del 8086. Incluye un bus de datos bidireccional de 8 bits (D_0 a D_7), que se utiliza para transferir datos a y desde los puertos de E/S e información

al registro de control. Las dos líneas de dirección especifican uno de los tres puertos de E/S o el registro de control. Una transferencia tiene lugar cuando se activa la línea *selección de módulo* conjuntamente con la línea de *lectura* o *escritura*. La línea de *reset* se emplea para inicializar el módulo.

El registro de control lo configura la CPU para controlar el modo de operación y definir las señales de control. En el modo de operación 0, los tres grupos de 8 líneas externas funcionan como 3 puertos de E/S de 8 bits cada uno. Cada puerto se puede seleccionar como puerto de entrada o de salida. En cualquier otro modo de operación los grupos *A* y *B* se emplean como puertos de E/S y las líneas del grupo *C* sirven como líneas de control para *A* y *B*. Las señales de control cumplen dos objetivos principales:

1) Establecer el protocolo de comunicación

2) Gestionar la petición de interrupciones

El protocolo de comunicación consiste en un simple mecanismo de temporización. Una línea de control la utiliza el periférico que envía la información como *línea de datos preparada*. De esta forma indica que el dato está ya sobre las líneas de datos de E/S. El receptor usa también una línea como *línea de reconocimiento*. Con esta línea señala que el dato ha sido leído y que por lo tanto se pueden borrar las líneas de datos. Otra línea puede representar una *línea de petición de interrupción* y estar conectada con el bus del sistema.

Como el 8255A es programable a través de su registro de control, se puede emplear para controlar una variedad de periféricos simples. La Figura 3.29 muestra su utilización en el caso de la pantalla y el teclado de un computador personal.

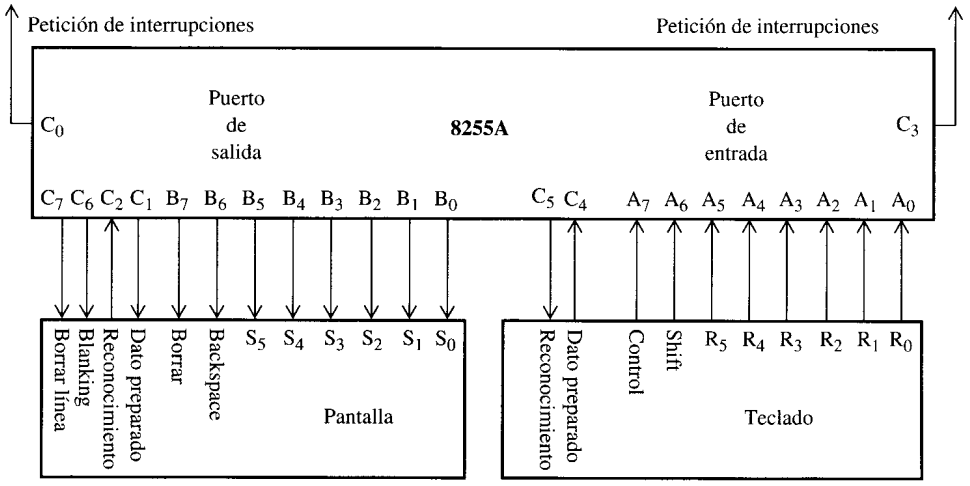


Figura 3.29: Interfaz teclado-pantalla con el 8255A

El teclado proporciona 8 bits de entrada al 8255A, dos de los cuales SHIFT y CONTROL tienen un significado especial para el programa que gestiona el teclado y que se ejecuta en la CPU. Sin embargo, esta interpretación es transparente al 8255A que simplemente acepta los 8 bits de datos y los presenta al bus de datos del sistema. Se disponen de dos líneas para realizar el protocolo de comunicación con el teclado.

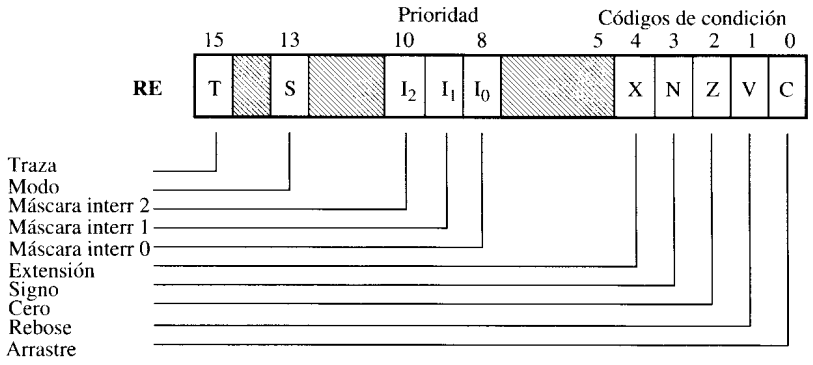
La pantalla está también conectada por un puerto de datos de 8 bits. Dos de los bits tienen un significado

especial que no le importa nada al 8255A. Además de las dos líneas para establecer el protocolo de comunicación se tienen otras dos líneas más, que permiten funciones de control adicionales.

3.4.9 Estructura de interrupciones del 68000

El 68000 tiene 8 niveles de prioridad para la gestión de las interrupciones. La prioridad con la que el procesador está operando en cualquier instante de tiempo se codifica en 3 bits de su *registro de estado* del 68000 (ver Figura 3.30). A estos 3 bits I_2, I_1 e I_0 ($I_{2\text{-}0}$) se les denomina *nivel de prioridad del procesador* o PPL (Processor Priority Level).

Los periféricos se conectan al 68000 utilizando una estructura análoga a la de la Figura 3.15a y donde a las peticiones de interrupción se les asignan prioridades en el rango 1 a 7. El 68000 posee 3 líneas de entrada para petición de interrupciones ($\overline{IPL2}$, $\overline{IPL1}$ e $\overline{IPL0}$) en las que cualquier dispositivo externo puede aplicar un número de 3 bits (0-7), que da el *nivel de prioridad de la petición de interrupción* o IPL (Interrupt-request Priority Level). El procesador aceptará una petición de interrupción solamente si el IPL dado por un periférico es numéricamente mayor que el PPL que tiene asignado en el registro de estado el programa que se está ejecutando. Hay una excepción a esta regla, las peticiones de interrupción de nivel 7 (IPL = 7) se atienden siempre (*interrupciones no enmascarables*).



T	Bit de traza	Si está a 1 la CPU entra en ejecución paso a paso
S	Bit de estado	Si está a 1 modo supervisor; a 0 modo usuario
I_2 I_1 I_0	Máscara de interrupción	Nivel de la máscara de interrupción
X	Bit de extensión	Se usa en operaciones de precisión múltiple
N	Bit de signo	Si está a 1 $\Rightarrow$ Resultado < 0; a 0 $\Rightarrow$ Resultado $\geq$ 0
Z	Bit de cero	Si está a 1 $\Rightarrow$ Resultado = 0
V	Bit de rebose	Se excede el máximo que se puede representar
C	Bit de arrastre	Se produce arrastre en el bit más significativo

Figura 3.30: Registro de estado (RE) del 68000

Por ejemplo, si PPL = 4, el procesador sólo aceptará peticiones de interrupción con IPL's de 5, 6 o 7. Si PPL = 0, el procesador aceptará IPL's desde 1 a 7. Como un IPL = 0 nunca es mayor que el valor del PPL, los periféricos utilizan este valor para señalar que no se tiene pendiente ninguna petición de interrupción. Esto implica que sólo existen 7 valores de IPL que indican verdaderas peticiones de interrupción .

El 68000, al igual que el 8086, utiliza interrupciones vectorizadas. Cuando acepta una petición de interrupción, la dirección de comienzo del programa de servicio de la interrupción la obtiene de un vector de interrupción almacenado en la memoria principal. Hay 256 vectores de interrupción (desde 0 a 255). Cada vector consta de 32 bits que constituyen la dirección de inicio requerida. Así, cuando un periférico solicita una interrupción, como respuesta a la señal de reconocimiento de la misma envía un número de 8 bits al procesador. Este número representa al vector de interrupción que se debe utilizar. De forma concisa la secuencia de acciones que realiza el 68000 para el tratamiento de interrupciones externas es la siguiente:

1) El periférico pone en $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$ e $\overline{\text{IPL0}}$ un valor distinto de 111 (recuérdese que las líneas IPL están complementadas y que por lo tanto 111 indica que no hay pendiente ninguna interrupción).
2) Si el campo $I_2I_1I_0$ del *RE* es mayor que el IPL enviado por el periférico, admite la interrupción.
3) Almacena el nivel de prioridad de la petición de interrupción (el valor de IPL) en el *RE*, es decir:

   $$I_2I_1I_0 = \overline{\text{IPL2}}\ \overline{\text{IPL1}}\ \overline{\text{IPL0}}$$

   De esta forma se evitan interrupciones con prioridades igual o menor.
4) El 68000 entra en el modo supervisor (S = 1 y T = 0).
5) Pone sobre las líneas de control $FC_{2\text{-}0}$ el valor 111 y sobre las líneas de dirección $A_3A_2A_1$ el valor de $I_2I_1I_0$.

   $FC_{2\text{-}0}$ son tres señales de estado del 68000 que indican el tipo de ciclo que está realizando el procesador, así cuando $FC_{2\text{-}0}$ = 111 se produce el reconocimiento de la interrupción.
6) Lee del bus un vector de 8 bits transmitido por el periférico.
7) Guarda en la pila el *RE* con los valores anteriores a las modificaciones y el contador de programa (*PC*).
8) Coloca en el contador de programa (PC) la doble palabra contenida en la dirección = 4 × vector y en consecuencia, el procesador bifurca al programa de servicio de la interrupción.
9) La última instrucción del programa de servicio debe ser un RTE, que restaura los valores del *PC* y del *RE*.

Sección 3-5

Acceso directo a memoria (DMA)

La E/S por interrupciones, aunque es más eficaz que la E/S controlada por programa, requiere también la intervención activa de la CPU para transferir datos entre la memoria y un periférico. En ambos casos, cualquier transferencia de datos debe recorrer un camino que pasa a través de la CPU. Estas dos formas de E/S sufren de dos desventajas inherentes:

1) La transferencia de datos está limitada por la velocidad con que la CPU puede comprobar y atender a un periférico.

2) La CPU está obligada a gestionar la transferencia de E/S; hay que ejecutar una serie de instrucciones durante cada operación de E/S (ver Figura 3.5 y Figura 3.13).

De hecho existe un cierto compromiso entre estas dos desventajas. Para ver esto, considérese la transferencia de un bloque de datos. Si se utiliza una E/S controlada por programa, la CPU está dedicada exclusivamente a la tarea de E/S y puede mover los datos con una relativa velocidad a costa de no hacer nada más. Por el contrario, si se emplea E/S por interrupción, se libera en cierta medida a la CPU a expensas de disminuir la velocidad de transferencia de E/S, ya que el programa de servicio de la interrupción puede contener muchas instrucciones ajenas a la transferencia del dato en si. Ambos métodos tienen pues un efecto adverso tanto sobre la actividad de la CPU como sobre la velocidad de transferencia de E/S.

Cuando se mueven grandes cantidades de datos, se necesita una técnica más eficaz: esta técnica es el *acceso directo a memoria* (Direct Memory Access).

3.5.1 Controlador de DMA

El DMA necesita un módulo adicional conectado al bus del sistema: el *controlador de DMA* (ver Figura 3.31) que es capaz de hacer las funciones asignadas a la CPU durante la transferencia de datos y asumir el control del sistema, especialmente de los buses de datos y de dirección. El controlador de DMA contiene, al igual que el controlador de E/S, un *registro de datos*, pero además tiene un *registro de dirección* y un *registro contador de palabras*. Estos registros permiten al controlador de DMA transferir datos desde (o hacia) una zona contigua de la memoria. El registro de dirección se utiliza para almacenar la dirección de la siguiente palabra que se va a transmitir y se incrementa de forma automática después de cada transferencia. El registro contador de palabras almacena el número de palabras que quedan por enviar y también se decrementa automáticamente después de cada transferencia. La unidad de control del DMA comprueba si el contenido del registro contador de palabras es 0 y cuando alcanza este valor para la transferencia, envía una señal de interrupción a la CPU para indicarle que la transferencia ha finalizado.

En resumen la técnica del DMA funciona de la manera siguiente: cuando la CPU desea leer o escribir un bloque de datos emite una orden al controlador de DMA enviándole la siguiente información:

a) Si la operación de E/S es de lectura o de escritura

b) La dirección del periférico

c) La posición de comienzo en memoria de donde hay que leer o donde hay que escribir

d) El número de palabras que se tienen que leer o escribir

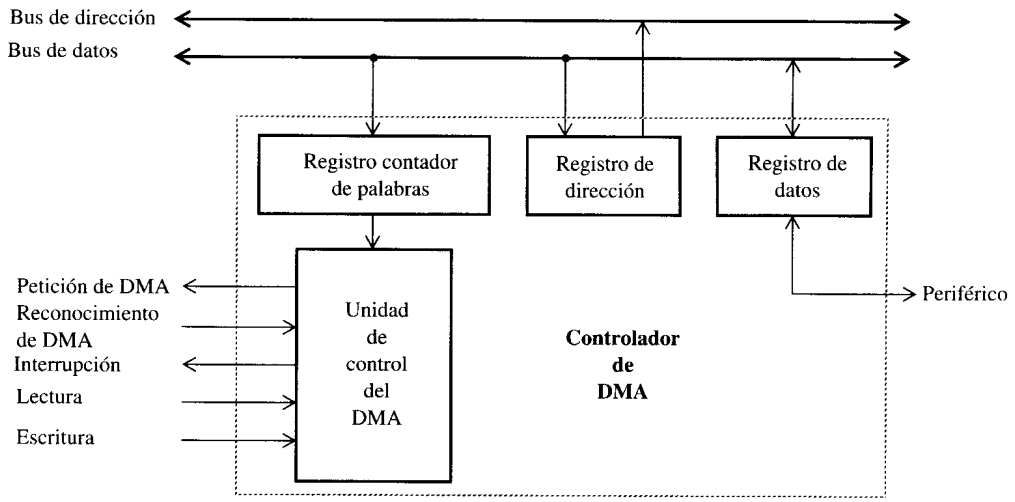


Figura 3.31: Diagrama de bloques de un controlador de DMA

A partir de este momento la CPU continúa realizando otra tarea. La CPU ha delegado esta operación de E/S en el controlador de DMA y es este módulo quien se encargará de ella. *El controlador de DMA transfiere directamente, palabra a palabra, el bloque completo de datos entre el periférico y la memoria, sin pasar por la CPU.* Cuando la transferencia finaliza, el controlador de DMA envía una señal de interrupción a la CPU. De esta forma la CPU únicamente participa al comienzo y al final de la transferencia (ver Figura 3.32).

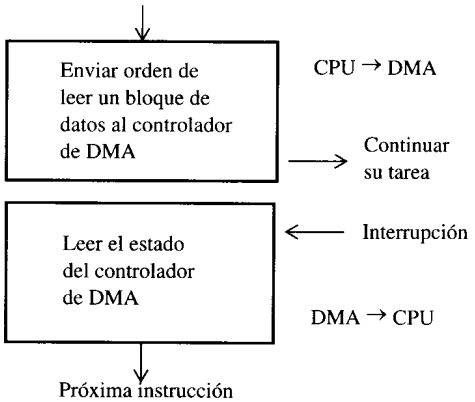


Figura 3.32: Lectura de un bloque de datos mediante DMA

3.5.2 Transferencia de datos mediante DMA

El controlador de DMA necesita tener el control del bus para poder transferir datos hacia (o desde) la memoria. Existen diferentes formas de obtener el control del bus que suponen distintos compromisos entre velocidad de transferencia y actividad de la CPU. El empleo de una estrategia en concreto dependerá de las prestaciones que se deseen y de las características del procesador que se utilice.

a) *Por ráfagas*: Cuando el DMA toma el control del bus no lo libera hasta haber transmitido el bloque de datos pedido tal como se observa en la Figura 3.33. Con este método se consigue la mayor velocidad de transferencia pero se tiene a la CPU inactiva durante períodos relativamente grandes. También se conoce como *parada del procesador.*

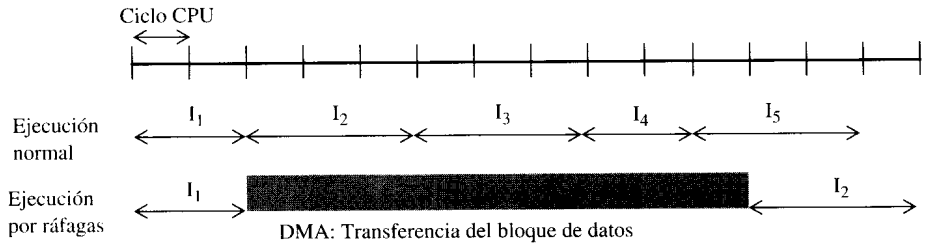


Figura 3.33: DMA por ráfagas

b) *Por robo de ciclos*: Cuando el DMA toma el control del bus lo retiene durante un sólo ciclo. Transmite una palabra y libera el bus. El proceso acaba cuando se ha transferido todo el bloque. Es la forma más usual de transferencia y en ella el DMA "roba" ciclos a la CPU. El robo de ciclos reduce la velocidad de transferencia y la interferencia del controlador de DMA sobre la actividad de la CPU (ver Figura 3.34). El método resulta de utilidad cuando se desea simultanear la ejecución de un programa con la recepción o transmisión de datos a velocidades moderadas (por ejemplo por vía serie). La CPU no tiene que ocuparse para nada de la operación y sigue ejecutando su programa, casi con la misma velocidad, mientras que de forma simultánea se efectúa la transferencia de datos.

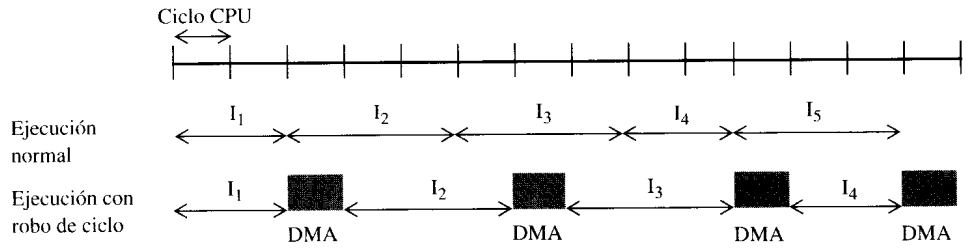


Figura 3.34: DMA por robo de ciclo

c) *DMA transparente*: Es posible eliminar completamente la interferencia entre el controlador de DMA y la CPU. Se consigue si se diseña el DMA de forma que solamente se roban ciclos cuando

la CPU no está utilizando el bus del sistema. La CPU no necesita usar el bus en todas las fases de la ejecución de una instrucción.

La ejecución del programa de la CPU no disminuye su velocidad, pero concurrentemente tiene lugar una transferencia por DMA. Análogamente al caso de robo de ciclos no se obtiene la ventaja de una velocidad de transferencia muy elevada propia del DMA. En la Figura 3.35 se ilustra el principio de este método de transferencia.

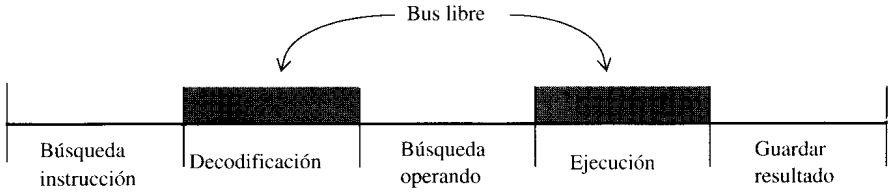


Figura 3.35: DMA transparente

La Figura 3.36 muestra donde la CPU puede suspender el ciclo de instrucción. En cada caso la CPU se para justamente antes de que se necesite utilizar el bus. El controlador de DMA entonces transfiere una palabra y devuelve el control a la CPU. Debe observarse que esto no es una interrupción; la CPU no guarda su contexto, como en el caso de la E/S por interrupción (el controlador de DMA no altera los registros de la CPU); lo que hace la CPU es parar su actividad durante un ciclo del bus. El efecto total es que la CPU ejecuta su programa más lentamente.

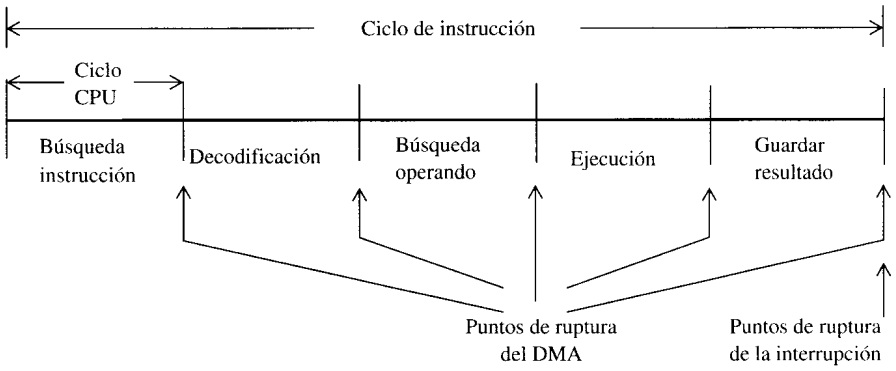


Figura 3.36: Puntos de ruptura del DMA y de la interrupción

En resumen, los pasos que se siguen en la transferencia mediante DMA son:

1) La CPU ejecuta tres instrucciones de E/S que cargan los registros de dirección de memoria, del periférico y el contador de palabras del controlador de DMA. El registro de dirección de memoria debe contener la dirección base de la zona de memoria principal que se va a utilizar en la transferencia de datos. El registro de dirección del periférico contiene la dirección del dispositivo del que hay que tomar o al que hay que enviar los datos. El registro contador de palabra almacena el número de palabras que se transfieren desde (o hacia) la memoria.

2) Cuando el controlador de DMA está preparado para transmitir o recibir datos, activa la línea de "petición del DMA" a la CPU. La CPU espera en el siguiente punto de ruptura del DMA, renuncia al control de los buses de datos y direcciones según el tipo de DMA y activa la línea de "reconocimiento de DMA".

   Las peticiones simultáneas de transferencia mediante DMA por parte de algunos controladores se resuelven utilizando las técnicas de control con prioridad de las interrupciones que se vieron en la sección anterior.

3) El controlador de DMA transfiere ahora directamente los datos a o desde la memoria principal por alguno de los métodos que se acaban de ver. Después de transferir una palabra, el registro de dirección y el registro contador de palabras del controlador se incrementa y decrementa respectivamente.

4) Si el contenido del registro contador de palabra no es 0, pero el periférico no está preparado para enviar o recibir el próximo lote de datos, el controlador de DMA devuelve el control a la CPU liberando el bus del sistema y desactivando la línea de petición de DMA. La CPU responde desactivando la línea de reconocimiento de DMA y continuando con su operación normal.

5) Si el contenido del contador de palabras es 0, el controlador de DMA renuncia al control del bus del sistema y envía una señal de interrupción a la CPU.

3.5.3 Configuración del DMA

El DMA, se puede configurar de diferentes formas. En las Figuras 3.37 a 3.39 se muestran algunas posibilidades. En el primer caso (ver Figura 3.37) todos los módulos comparten el mismo bus del sistema. El controlador de DMA, que actúa como un sustituto de la CPU, utiliza E/S controlada por programa para intercambiar datos entre la memoria y un periférico a través del controlador de DMA.

Esta configuración, aunque puede ser muy económica, es claramente poco eficaz, ya que cada transferencia de una palabra consume dos ciclos del bus igual que con la E/S controlada por programa.

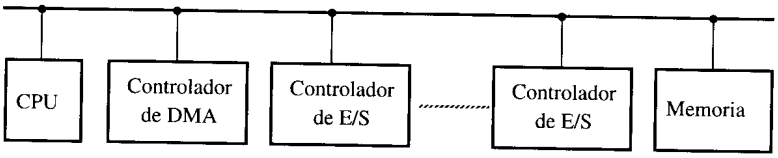


Figura 3.37: Bus único con DMA conectado

Se puede reducir sustancialmente el número de ciclos de bus necesitados integrando las funciones de DMA y E/S. Tal como se muestra en la Figura 3.38, esto significa que hay un camino entre el controlador de DMA y uno o más controladores de E/S que no incluyen al bus del sistema. La lógica del DMA puede ser una parte de un controlador de E/S o puede ser un módulo independiente que controla a uno o más controladores de E/S.

El concepto anterior se puede generalizar si se utiliza un bus de E/S para conectar los controladores de E/S al controlador de DMA (ver Figura 3.39). Este método reduce a una el número de interfaces de E/S en el controlador de DMA y proporciona una configuración fácilmente ampliable.

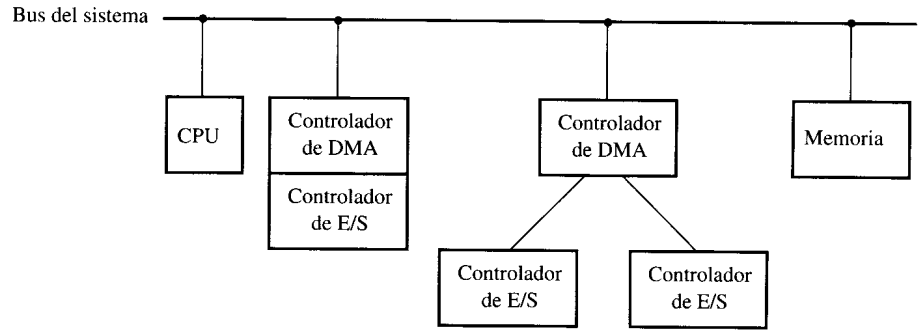


Figura 3.38: Bus único con DMA integrado

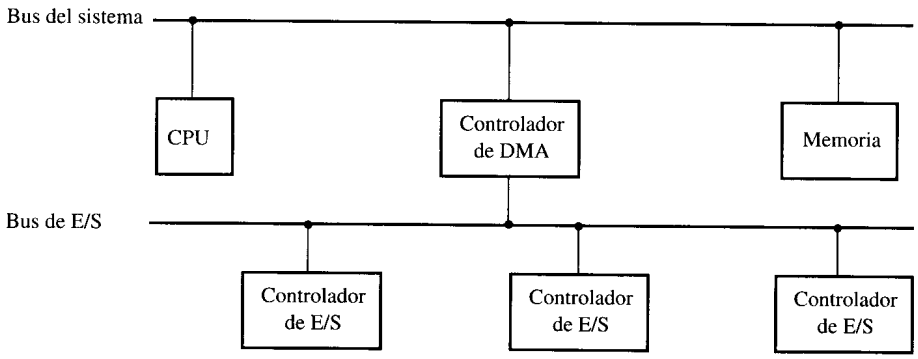


Figura 3.39: Controlador de DMA conectado al bus de E/S

En todos los casos el bus del sistema, que el controlador de DMA comparte con la CPU y la memoria, lo utiliza únicamente para intercambiar datos con la memoria. Sin embargo el flujo de datos entre el DMA y los controladores de E/S tienen lugar fuera del bus del sistema sólo en las dos últimas situaciones.

Sección 3-6

Procesador de E/S (PE/S)

Con el paso del tiempo el computador ha ido modificándose hacia estructuras más complejas al utilizar componentes individuales más sofisticados. Donde la adaptación ha sido más evidente es en la función de E/S. A lo largo del tema se ha visto ya parte de esa evolución que se puede resumir de la forma siguiente:

1) La CPU controla directamente al periférico. Solución que se observa en sistemas simples controlados por un microprocesador y dedicados a una única tarea.

2) Se incorpora un controlador de E/S. La CPU utiliza E/S controlada por programa sin interrupciones. Con este paso, la CPU se libera en cierta medida de los detalles específicos de la interfaz con el periférico.

3) Se utiliza la misma configuración que en el paso 2), pero ahora se emplean interrupciones. La CPU no malgasta su tiempo esperando a que se realice una operación de E/S. Esto aumenta el rendimiento global del sistema.

4) Al controlador de E/S se le permite el acceso directo a memoria a través de DMA. Se pueden transferir bloques de datos a o desde memoria sin intervención de la CPU, excepto al comienzo y al final de la transferencia.

5) Se potencia al controlador de E/S para convertirlo en un procesador con un conjunto de instrucciones especializadas en operaciones de E/S. La CPU dirige al procesador de E/S (PE/S) para que ejecute un programa de E/S que está residente en la memoria. El PE/S busca y ejecuta ese programa sin la intervención de la CPU y permite a la CPU especificar una secuencia de actividades de E/S que sólo se interrumpe cuando se ha ejecutado la secuencia completa.

6) El procesador de E/S tiene una memoria local y se puede considerar que es un computador. Con esta arquitectura se consigue controlar un gran número de periféricos con una intervención mínima de la CPU. Tal configuración se emplea normalmente para controlar la comunicación con terminales interactivos. El PE/S se encarga de la mayor parte de las tareas que son necesarias.

Cuando se avanza a lo largo de este camino se observa que cada vez se realizan más funciones de E/S sin intervención directa de la CPU. En los pasos 5) y 6) se produce un cambio fundamental con la introducción del concepto de *un controlador de E/S capaz de ejecutar un programa*. Los controladores de E/S de los pasos 5) y 6) se suelen denominar *canal de E/S* y *procesador de E/S* (PE/S) respectivamente. No obstante los dos términos se emplean de forma indistinta para ambas situaciones. En lo que sigue se utilizará el término PE/S.

3.6.1 Características de los PE/S

El PE/S representa una extensión del concepto DMA. Un PE/S tiene la capacidad de ejecutar instrucciones de E/S, lo que le da un control completo sobre dicha operación. En los computadores que incluyen PE/S, la CPU no ejecuta las instrucciones de E/S, éstas se almacenan en memoria principal para ser ejecutadas por un PE/S. Así, la CPU inicia una transferencia de E/S al dar una orden al PE/S para que ejecute un programa en memoria. El programa especifica entre otras cosas las siguientes:

a) El periférico (o periféricos) que interviene en la operación de E/S.

b) La zona de memoria utilizada en la transferencia.

c) Las prioridades.

d) Qué acciones hay que efectuar si se producen ciertas condiciones de error durante la transferencia.

El PE/S sigue estas instrucciones y controla la transferencia de datos. En la Figura 3.40 se muestra el diagrama de bloques de un computador con dos procesadores. La memoria ocupa una posición central y se puede comunicar con cada procesador por DMA. La CPU es responsable de procesar los datos que se necesitan en la solución de una determinada tarea computacional. El PE/S proporciona un camino para la transferencia de datos entre los diferentes periféricos y la memoria. A la CPU se le asigna normalmente la tarea de iniciar el programa de E/S. A partir de ese momento el PE/S opera independientemente de la CPU y continúa transfiriendo datos entre el dispositivo externo y la memoria.

Los formatos de los datos de los diferentes periféricos difieren a menudo de los de la memoria y la CPU. El PE/S debe estructurar datos de muy diversas fuentes. Por ejemplo, puede ser necesario tomar 4 bytes de un dispositivo de entrada y empaquetarlos en una palabra de 32 bits antes de transferirla a memoria. Los datos se reunen en el PE/S con la velocidad que impone el dispositivo externo mientras la CPU está ejecutando su propio programa. Después de su ensamblado en una palabra de memoria, el dato se transfiere directamente desde el PE/S a memoria robando un ciclo de memoria a la CPU. Análogamente una palabra de salida transferida desde memoria al PE/S se dirige desde este al dispositivo de salida a la velocidad que marca el periférico.

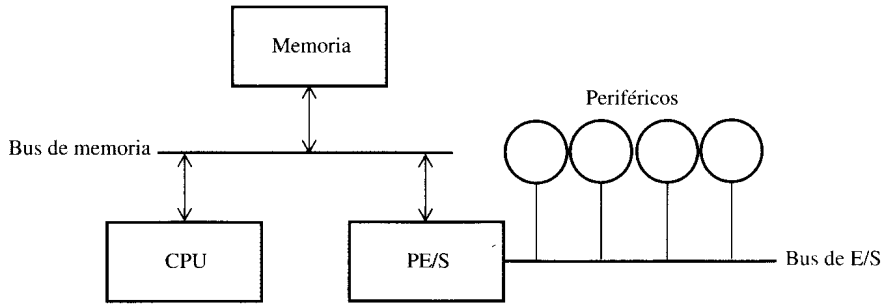


Figura 3.40: Diagrama de bloques de un computador con un PE/S

La comunicación entre el PE/S y los dispositivos que tiene conectados es similar al método de transferencia de E/S controlada por programa mientras que la comunicación con la memoria es análoga a la E/S controlada por DMA. La forma en que se comunica entre si la CPU y el PE/S depende del nivel de

sofisticación del sistema. En los grandes sistemas cada procesador es independiente de todos los otros y puede iniciar una operación. Normalmente la CPU es el procesador maestro mientras que los PE/S´s son los procesadores esclavos. A la CPU se le asigna la tarea de iniciar todas las operaciones, pero las instrucciones de E/S se ejecutan en un PE/S. Las instrucciones de la CPU proporcionan operaciones para iniciar una transferencia de E/S y también para comprobar las condiciones del estado de dicha transferencia que se pueden necesitar para tomar alguna decisión sobre diferentes actividades de la operación de E/S. El PE/S a su vez solicita la atención de la CPU por medio de una interrupción. También responde a las peticiones de la CPU colocando una palabra de estado en una posición determinada de memoria o en un puerto de E/S para que pueda ser examinada posteriormente por un programa que ejecuta la CPU. Cuando se desea una operación de E/S, la CPU informa al PE/S donde puede encontrar el programa de E/S y a continuación deja los detalles de la transferencia al PE/S.

Las instrucciones que se leen de la memoria por un PE/S algunas veces se denominan *órdenes* para diferenciarlas de las instrucciones que se transmiten a la CPU. Una instrucción y una orden tienen funciones análogas. Las órdenes se preparan por los programadores y se almacenan en memoria.

La comunicación entre la CPU y el PE/S puede tomar diferentes formas dependiendo del computador en particular que se utilice. En la mayoría de los casos la memoria actúa como un centro de envío de mensajes, donde cada procesador deja información al otro. Para apreciar la operación de un PE/S típico, se resume el método que emplean para comunicarse entre si la CPU y el PE/S. Este ejemplo simplificado omite muchos detalles operativos y se centra en proporcionar una visión panorámica de los conceptos básicos. En la Figura 3.41 se muestra en un diagrama la secuencia de operaciones.

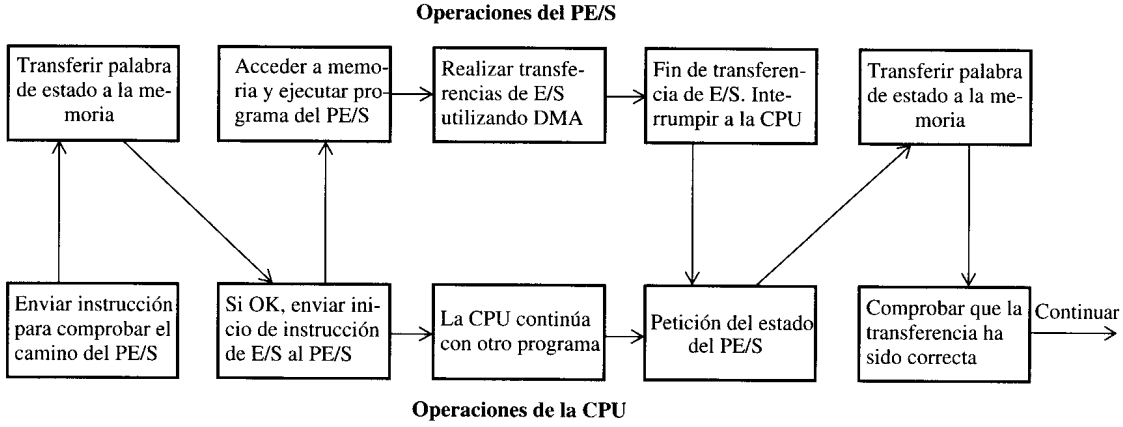


Figura 3.41: Comunicación CPU-PE/S

3.6.2 Clasificación de los PE/S

Los dos tipos de PE/S más comunes que se suelen emplear son los siguientes:

1) *Canal selector*

2) *Canal multiplexor*

Un canal selector controla múltiples dispositivos de alta velocidad. *En cualquier instante de tiempo está dedicado a la transferencia de datos con uno sólo de estos dispositivos.*

De esta forma el PE/S selecciona un dispositivo y efectúa la transferencia de datos (ver Figura 3.42). Cada dispositivo, o un pequeño número de ellos, está manejado por un controlador de E/S del tipo que se ha visto anteriormente. El PE/S sustituye a la CPU para supervisar el funcionamiento de los controladores de E/S.

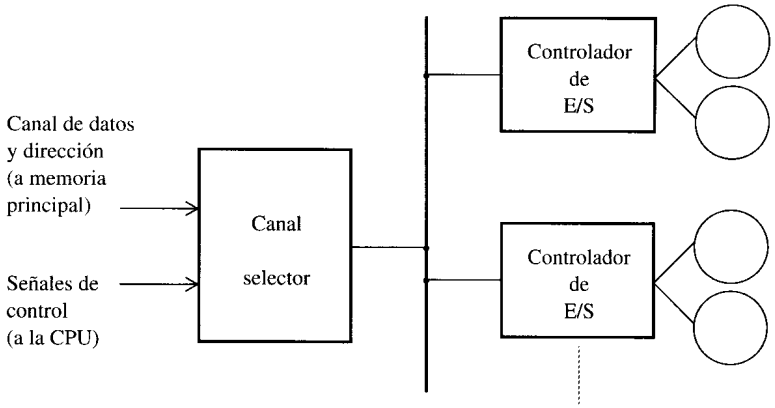


Figura 3.42: Arquitectura de un PE/S tipo selector

Un canal multiplexor puede controlar de forma simultánea operaciones de E/S con múltiples dispositivos (ver Figura 3.43). Para periféricos de baja velocidad, un multiplexor orientado a la "transferencia de bytes" acepta o transmite caracteres de la forma más rápida posible a o desde los dispositivos con los que está conectado. Por ejemplo, el flujo resultante de caracteres desde tres periféricos con diferentes velocidades y flujos individuales de:

$$a_1\, a_2\, a_3\, a_4 \ldots \qquad b_1\, b_2 b_3\, b_4 \ldots \qquad c_1\, c_2\, c_3\, c_4 \ldots$$

podría ser:

$$a_1\, b_1 c_1\, a_2\, c_2\, a_3\, b_2\, c_3\, a_4 \ldots$$

Para dispositivos de alta velocidad, un multiplexor orientado a la "transferencia de bloques", alterna bloques de datos de algunos dispositivos.

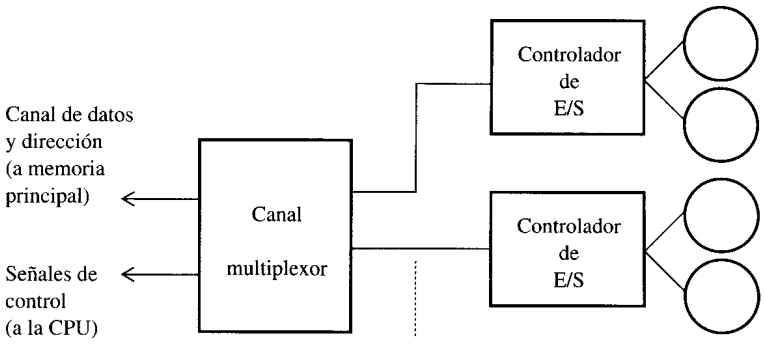


Figura 3.43: Arquitectura de un PE/S tipo multiplexor

Sección 3-7

Conclusiones

La función básica de un computador se realiza únicamente entre dos de sus componentes, la CPU y la memoria principal. La CPU busca las instrucciones y los datos de la memoria principal, los procesa y almacena los resultados otra vez en la memoria. Los otros elementos de un computador se pueden considerar en un sentido muy general como "unidades de E/S", puesto que su objetivo es transferir información entre la memoria principal o la CPU y el mundo exterior. La unidad de E/S incluye tanto los dispositivos de E/S como los controladores de E/S, que son los encargados del control de uno o más periféricos y del intercambio de datos entre dichos periféricos con la memoria principal o con los registros de la CPU.

Las unidades de E/S se clasifican según el grado de participación que toma la CPU en la ejecución de las operaciones de E/S. Una operación de E/S significa una transferencia de datos entre un periférico y la CPU o entre un periférico y la memoria principal. Si las operaciones de E/S están controladas completamente por la CPU (es decir, la CPU ejecuta los programas que inicializan, controlan y finalizan las operaciones de E/S) se dice que el computador está utilizando *E/S controlada por programa*. La E/S controlada por programa la poseen todos los computadores y requiere pocos elementos, pero hace que la CPU pase una gran parte de su tiempo realizando funciones de control de E/S relativamente triviales. Una de estas funciones es comprobar el estado de los periféricos para determinar si necesitan que se les atienda.

El periférico o el controlador de E/S también pueden tener la capacidad de activar una determinada línea de control para solicitar un servicio de la CPU: la ejecución de un programa específico que atiende al periférico. Este tipo de petición se llama una *interrupción*, que libera a la CPU de la tarea de comprobar periódicamente el estado del periférico. La interrupción provoca que la CPU conmute de programa, guardando el contexto del programa interrumpido y transfiriendo el control a un programa que maneja la interrupción. Cuando la interrupción ha sido atendida, la CPU puede continuar la ejecución del programa interrumpido.

Con un incremento moderado de la lógica asociada con el periférico se puede lograr transferir un bloque de información a o desde la memoria principal sin la intervención directa de la CPU. Esto requiere que los periféricos o los controladores de E/S sean capaces de generar direcciones de memoria y transferir datos a o desde el bus del sistema. La CPU es todavía la responsable de iniciar la transferencia de cada bloque. A partir de este punto el controlador de E/S realiza toda la operación sin que la CPU tenga que ejecutar ningún otro programa. La CPU y el controlador de E/S interaccionan sólo cuando la CPU debe dar el control del bus del sistema al controlador de E/S, como respuesta a una petición de este último. Este tipo de capacidad es lo que se conoce como *Acceso Directo a Memoria* (DMA). La mayoría de los computadores de hoy día tienen la posibilidad de interrupción y de DMA. Un controlador de DMA posee un control parcial de la operación de E/S. La CPU puede liberarse totalmente si se introduce en el computador un *Procesador de E/S* (PE/S) o *canal*. Análogamente al DMA, el PE/S tiene acceso a la memoria principal y puede interrumpir a la CPU, sin embargo puede usar un repertorio de instrucciones distinto al de la CPU (ya que está orientado a operaciones de E/S).

Sección 3-8

Problemas

1) En las interrupciones vectorizadas ¿por qué coloca el periférico el vector de interrupción sobre las líneas de datos y no sobre las líneas de dirección? (ver Figura 3-16)

2) En la práctica totalidad de los sistemas que incluyen controladores de DMA éstos poseen una prioridad más elevada que la CPU en los accesos a memoria principal. ¿Por qué?

3) Un controlador de DMA está transmitiendo, mediante robo de ciclos, caracteres a memoria desde un periférico a una velocidad de 19200 bps (bits/seg). Por su parte la CPU realiza la búsqueda de las instrucciones con una velocidad de 2 millones de instrucciones/seg (2 MIP). ¿En qué porcentaje se reduce la velocidad del procesador debido al controlador de DMA?

4) Una de las ventajas de usar E/S localizada en memoria frente a E/S aislada es la de permitir una programación más flexible, debido a poder emplear todo el repertorio de instrucciones máquinas con referencia a memoria. Por otro lado, una desventaja es la de gastar espacio de direcciones de memoria quitándoselo a instrucciones y datos. Explicar otras dos ventajas y desventajas.

5) Se dispone de un computador que funciona a una velocidad de 10^7 ciclos/seg y en el que, en promedio, una instrucción emplea 4 ciclos máquina y cada operación de lectura o escritura de memoria tarda 1 ciclo máquina. Determinar la máxima velocidad de transferencia de datos (en palabras/segundo) suponiendo que el computador utiliza:

   a) E/S controlada por programa y se emplean 3 instrucciones en transferir cada palabra.

   b) Un sistema de DMA.

6) Con referencia a la configuración de DMA de la Figura 3.44 explicar lo siguiente:

   a) ¿Puede la memoria auxiliar transferir datos a la memoria principal al mismo tiempo que la CPU lee datos de esa misma memoria?

   b) ¿Puede la CPU recibir datos del controlador de E/S al mismo tiempo que la memoria auxiliar recibe datos de la memoria principal?

   c) ¿Es posible que el controlador de E/S transmita datos a la memoria principal mientras que la CPU ejecuta una operación aritmética?

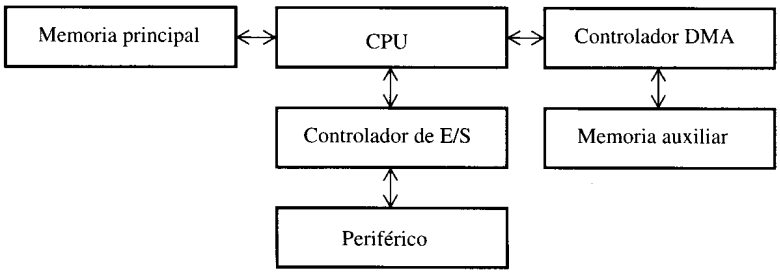


Figura 3.44: Configuración con DMA

7) En la Figura 3.44 del problema anterior, ¿cuál de los siguientes módulos comienza una operación de DMA?:

a) ¿La memoria principal?

b) ¿La CPU?

c) ¿La memoria auxiliar?

8) Un computador de 32 bits dispone de dos canales selectores y un canal multiplexor. Cada canal selector controla dos unidades de discos y dos unidades de cinta. Al canal multiplexor están conectados dos impresoras, dos lectoras de tarjetas y 10 pantallas. Para los diferentes periféricos se suponen las siguientes velocidades de transferencias:

Unidad de disco	800 Kbytes/seg
Unidad de cinta magnética	200 Kbytes/seg
Impresora	8Kbytes/seg
Lectora de tarjeta	1,5 Kbytes/seg
Pantalla	1Kbyte/seg

Estimar la máxima velocidad de transferencia de E/S agregada en este sistema.

9) ¿Qué sucede en la interrupción por encadenamiento de la Figura 3.19, cuando el periférico 1 solicita una interrupción después que el periférico 2 ha enviado una petición de interrupción a la CPU, pero antes de que ésta responda con la señal de reconocimiento de interrupción?

10) Se dispone de un computador que no posee ningún hardware de interrupciones, pero que puede recibir solicitud de interrupción de diferentes periféricos. Cualquier petición de interrupción provoca el almacenamiento de la dirección de retorno y la bifurcación a un programa de servicio común a todas las interrupciones. Explicar cómo se puede establecer un esquema de prioridades en el programa de servicio de las interrupciones.

11) Obtener la tabla de verdad de un codificador de prioridad 8 × 3. Suponer que las tres salidas *a, b* y *c* se utilizan para proporcionar un vector de dirección de la forma 110*abc*00. Obtener los 8 vectores de dirección comenzando por el de mayor prioridad.

12) Un computador dispone de una única línea de petición de interrupción IRQ y utiliza un mecanismo de interrupciones mediante encadenamiento. Cada uno de sus dispositivos periféricos D_i posee un circuito de interfaz que gestiona sus señales de encadenamiento. Este circuito de interfaz recibe una señal de aceptación de interrupción $INTA_i$ desde el interfaz del dispositivo D_{i-1} y genera $INTA_{i+1}$ para el dispositivo D_{i+1}.

El dispositivo D_i se comunica con su circuito de interfaz mediante las señales PI_i y RI_i. El dispositivo requiere el servicio activando la línea de petición PI_i. Entonces el interfaz activa la línea IRQ_i y cuando recibe $INTA_i$ activa la línea RI_i, que indica que la comunicación ha sido aceptada, desactiva IRQ_i y saca un vector de interrupción $dint_i$, identificativo del dispositivo, al bus de dirección (ver Figura 3.45). Implementar el circuito de interfaz de cada dispositivo D_i, utilizando biestables tipo D y los componentes combinacionales necesarios.

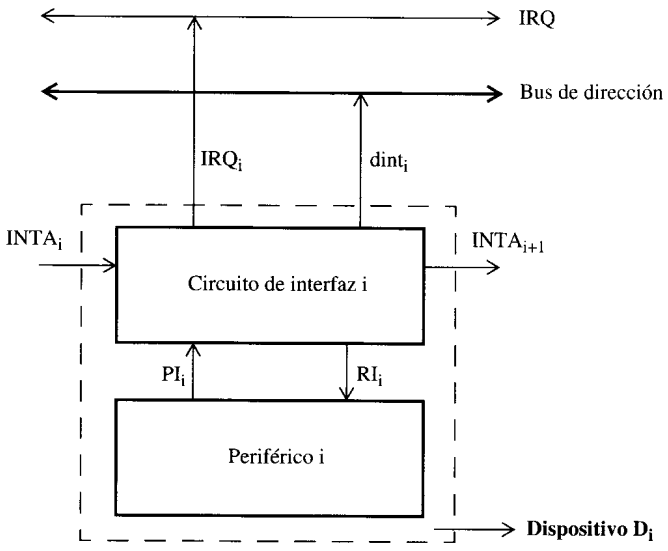


Figura 3.45: Conexión de un periférico al resto del sistema mediante su circuito de interfaz

13) Indicar bajo qué circunstacias el mecanismo de encadenamiento propuesto en el problema anterior puede dar lugar a que un periférico al que no se le ha concedido una interrupción pueda colocar indebidamente su vector de interrupción en el bus de dirección.

14) Un computador tiene tres líneas de petición de interrupción PI_i (i = 1, 2, 3) y tres líneas de reconocimiento asociadas RI_i (i = 1, 2, 3). Cuando se reciben peticiones de interrupción simultáneas por más de una línea sólo se reconoce la más prioritaria (la máxima prioridad la tiene $INTR_1$ y la mínima $INTR_3$). La UCP dispone únicamente de una línea de petición PI y una línea de reconocimiento RI.

a) Obtener la tabla de verdad de las salidas RI_1, RI_2 y RI_3.

b) Obtener un circuito lógico que implemente este esquema de prioridades.

15) Sea un microcomputador (μC) con un bus de datos de 8 bits que cuenta con una sóla entrada de petición de interrupción, *PI*, y la correspondiente salida de reconocimiento, *RI*. Se desea dotar a este μC de 4 niveles de petición de interrrupción en cada uno de los cuales pueden interrumpir 4 periféricos. Diseñar la lógica de control de interrupciones que es necesario añadir al sistema de modo que:

1) Se resuelvan las prioridades entre los distintos niveles.
2) Se resuelvan las prioridades entre los dispositivos de un mismo nivel.
3) El dispositivo que interrumpe pueda identificarse mediante un vector de 8 bits.

16) Sea un μC con 8 líneas de datos, 16 líneas de direcciones, una línea de petición de interrupción *PI* y una línea de reconocimiento *RI*. Diseñar un controlador de interrupciones para dotar a este sistema de 4 niveles vectorizados enmascarables selectivamente, de tal forma que:

1) Los registros que almacenan los vectores forman parte del controlador de interrupciones y pueden leerse y escribirse desde la CPU. Estos registros ocupan las direcciones hexadecimales F000-F003.
2) Las mismas consideraciones para el registro de máscara. Su dirección es F004.

17) Rehacer el problema anterior considerando que el controlador de interrupciones debe ser encadenable, es decir, que la utilización de n controladores permita disponer de $4n$ niveles de interrupción.

18) Se desea diseñar una unidad de control DMA para 4 disquetes que cumpla los siguientes requisitos:

- Se comunicará con la CPU mediante E/S programada.
- Admitirá simultáneamente una petición para cada uno de los disquetes.
- Resolverá los conflictos cuando haya peticiones para más de un disquete.

Se pide diseñar:

a) Banco de registros necesarios, indicando su función.
b) Decodificación para el acceso a estos registros desde la CPU, suponiendo que ocupan las direcciones F000H y siguientes.
c) Lógica de prioridades para resolver los conflictos de acceso.
d) Diagrama de flujo del funcionamiento de la unidad.

Suponer un ancho de palabra de 8 bits y un bus de direcciones de 16 bits.

19) Diseñar un controlador de DMA (CDMA) que opere con el microprocesador de la Figura 3.46. El microporcesador es capaz de direccionar una memoria de $64K$ palabras de 8 bits y hace uso de un bus síncrono. Los ciclos máquina de este microprocesador comienzan con el flanco de bajada de la señal de reloj. Las transferencias se realizarán únicamente en modo ráfaga. Se supone que el periférco es lo suficientemente rápido como para tomar o colocar un dato en el bus por cada ciclo de reloj, y que esto se hace coincidiendo con la nivel alto de la señal de reloj CK. También se supone que el periférico es capaz de realizar las transferencias con el bus directamente, sin necesidad de pasar por el registro de

datos del CDMA. Las líneas de arbitraje del bus son *HALT* y *BA*, y la forma en que opera el DMA una vez inicializado es la siguiente:

a) Pide la utilización del bus mediante *HALT* (parada).

b) La CPU le cede el bus en el ciclo siguiente y activa *BA* (bus disponible).

c) El DMA controla el bus realizando la transferencia del bloque.

d) El DMA genera una petición de interrupción para indicar que el bloque ha sido completamente transferido.

No considerar el proceso de inicialización del DMA. (Las líneas *LEC* y *ESC* indican lectura y escritura, respectivamente.)

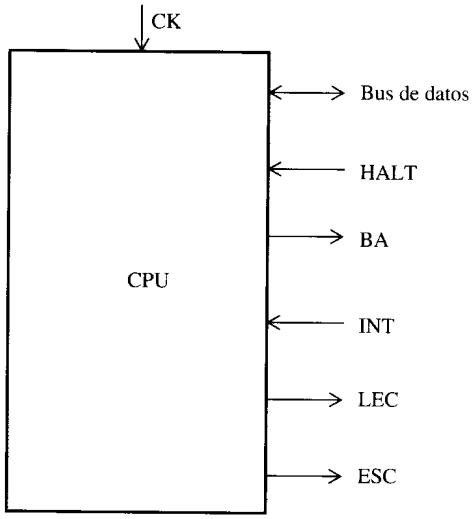


Figura 3.46: Diagrama de bloques del microprocesador

20) Completar el problema anterior añadiendo los mecanismos necesarios para la inicialización del DMA (escritura en los registros de trabajo). Suponer que los registros DIR (registro de dirección), CONT (registro contador de palabras) y SENT ocupan las direcciones FFFD - FFFF, respectivamente.

21) Se desean transferir 512 palabras desde una unidad de disco a una zona de la memoria que comienza en la dirección 2460. La transferencia es por medio de un controlador de DMA.

a) Dar los valores iniciales que la CPU debe transferir al controlador de DMA.

b) Expresar paso a paso las acciones que tienen lugar durante la transferencia de las dos primeras palabras.

22) Un controlador de E/S controla un periférico capaz de hacer transferencias vía DMA en un sistema de E/S localizada en memoria. El sistema dispone de una línea de lectura/escritura, $R/\overline{W}$, con el siguiente significado:

$R/\overline{W} = 0 \rightarrow$ escritura, $R/\overline{W} = 1 \rightarrow$ lectura.

El controlador de E/S debe enviar datos hacia el bus en las siguientes situaciones:

a) Cuando la CPU realiza una operación de lectura de uno de sus registros internos ($R/\overline{W} = 1$).

b) Cuando el controlador de DMA ordena la ejecución de una transferencia desde el periférico hacia la memoria (El controlador de DMA hará $R/\overline{W} = 0$ para escribir el dato en memoria).

Diseñar el circuito de control de las puertas triestado que comunican el controlador de E/S con el bus de datos.

23) Una planta industrial usa sensores para controlar variables tales como temperatura, presión, etc. La salida de cada sensor es un conmutador ON/OFF que se activa cuando la variable que vigila rebasa un cierto límite. Se desea conectar 8 de tales sensores a un microcomputador basado en un microprocesador idéntico al del problema 21. Diseñar el interfaz apropiado para que el estado de los 8 sensores pueda leerse desde la CPU como el contenido de la celda de memoria cuya dirección es:

a) F004.

b) Una cualquiera comprendida entre A000 y BFFF.

24) ¿Cuál es la diferencia esencial entre un controlador de DMA y un PE/S?

Tema 4

Unidad aritmético-lógica

La *unidad aritmético-lógica* (ALU[1]) es la parte del computador donde se efectúan las operaciones aritméticas y lógicas sobre los datos. Las otras unidades del computador (unidad de control, memoria y unidad de E/S) son las encargadas de suministrar datos a la entrada de la ALU y recibirlos nuevamente una vez procesados. La Figura 4.1 muestra en términos generales como se interconecta la ALU con el resto de la CPU.

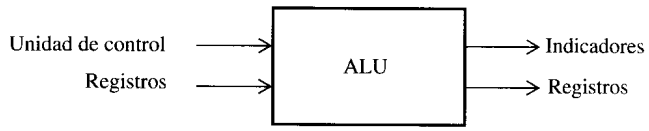


Figura 4.1: Entradas y salidas de la ALU

Los datos llegan a la ALU a través de registros y los resultados que se generan también se almacenan en registros. Estos registros son memorias temporales dentro de la CPU que se conectan mediante el bus de datos con la ALU. Cuando la ALU finaliza una operación, activa determinados indicadores que pueden ser utilizados por la unidad de control. La unidad de control envía señales que controlan las operaciones y el movimiento de datos de entrada y salida de la ALU. En este tema se estudian los algoritmos y los circuitos asociados que realizan las cuatro operaciones aritméticas básicas, tanto en coma fija como en coma flotante. Un número en coma flotante está constituido por un par de números en coma fija, la mantisa m y el exponente e y se utiliza para representar números de la forma $m \times B^e$, donde B es la base que está implícita. La representación en coma flotante aumenta el rango de los números que se pueden expresar para una longitud de palabra dada, aunque requieren circuitos aritméticos mucho más complejos que cuando se emplea coma fija. Con el fin de proporcionar una representación única para cada número en coma flotante se realiza un proceso de normalización.

También se analizan las operaciones de desplazamiento y de comparación.

---

1. A lo largo de todo el tema se utilizará el término ALU que es el acrónimo de (*Arithmetic-Logical Unit*) para designar a la unidad aritmético-lógica.

Sección 4-1

Sumadores binarios

Un sumador binario se puede considerar como un conversor de código que recibe a la entrada dos números binarios x e y de n bits cada uno (los operandos)

$$x = x_{n-1}\, x_{n-2} \ldots\ldots x_1\, x_0$$

$$y = y_{n-1}\, y_{n-2} \ldots\ldots y_1\, y_0$$

y produce una salida s de $n + 1$ bits que es la suma de los operandos (ver Figura 4.2).

$$s = s_n\, s_{n-1} \ldots\ldots s_1\, s_0$$

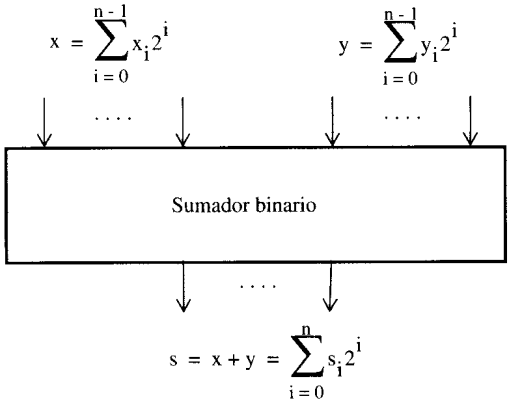


Figura 4.2: Sumador binario de n bits

En la realización de un sumador binario aparece un compromiso entre velocidad y coste. En principio, la forma más rápida de realizar cualquier función lógica, cuando el valor de sus salidas está únicamente determinado por el valor presente de sus entradas, es mediante un circuito combinacional de dos niveles (AND-OR). Se observa en la Tabla 4.1 que el bit s_j de la suma (j = 0, 1, ..., n) es función de las $2(j+1)$ variables $x_i, y_i (i = 0, 1, ..., j)$.

Por lo tanto, en una realización canónica que incluyese todos los términos productos, el bit s_j de la suma requeriría $2^{2(j+1)}$ puertas AND de $2(j+1)$ entradas y una puerta OR de $2^{2(j+1)}$ entradas. Este número de puertas resulta excesivo incluso para valores moderados de n, por lo que es inviable esta solución y es preciso desarrollar otros métodos que, aunque más lentos, sean menos costosos.

s_0	=	$s_0\,(x_0, y_0)$
s_1	=	$s_1\,(x_1, y_1, x_0, y_0)$
.....		
s_{n-1}	=	$s_{n-1}\,(x_{n-1}, y_{n-1}, \ldots\ldots, x_1, y_1, x_0, y_0)$
s_n	=	$s_n\,(x_{n-1}, y_{n-1}, \ldots\ldots, x_1, y_1, x_0, y_0)$

Tabla 4.1: Dependencia de los bits de suma de los bits de los operandos

La estrategia básica que se emplea para sumar dos números de n bits es dividir el problema en un conjunto de tareas más sencillas, de forma tal que una vez resueltas se tenga, por simple composición, la solución del problema original. La partición más usual es transformar el problema de *suma de dos números de n bits* en *n problemas idénticos de suma de dos números de 1 bit.*

Cada una de estas n sumas más sencillas necesitan información de los sumadores de las etapas previas (*arrastres*). Estas señales de arrastre (denominadas también de *acarreo*) que se producen en los sumadores elementales de orden inferior (de los bits menos significativos al que se considera), son el origen de las dificultades que surgen en la operación de suma.

A continuación se analizan algunas realizaciones del circuito lógico básico para la suma de dos números de 1 bit (semisumador), su generalización cuando se considera el arrastre de la etapa previa (sumador binario completo) y como se efectúa la suma de dos números de n bits interconectando adecuadamente sumadores binarios completos. En la Figura 4.3 se muestra el esquema de la *representación de puntos* de la suma de dos números binarios de n bits. Consta de los siguientes elementos:

a) Una matriz de puntos de 2 filas × n columnas, donde cada uno de los puntos representa un bit individual de los dos números (x e y) a sumar.

b) Una linea horizontal que separa las representaciones de la entrada y de la salida.

c) Un vector fila de puntos de $n + 1$ columnas que representan los bits de la suma (s).

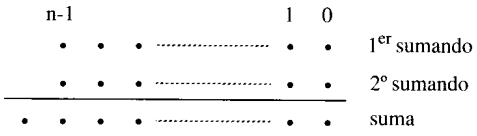


Figura 4.3: Representación de puntos de un sumador binario de n bits

4.1.1 Semisumador binario (SSB)

Es el circuito aritmético más sencillo. Consta de dos entradas binarias (x e y) y dos salidas; una de ellas (s) es el resultado de la "suma módulo 2" de las dos entradas y la otra (c) es el denominado "arrastre" (o acarreo) que indica si el resultado de la suma es igual a 2 (ver Tabla 4.2).

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabla 4.2: Tabla de verdad del SSB

En la Figura 4.4 se da el diagrama de un semisumador binario (SSB) y su representación de puntos .

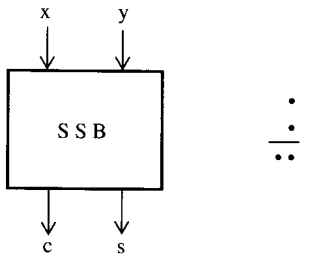


Figura 4.4: Diagrama de bloques y representación de puntos de un SSB

Las funciones lógicas de s y c, deducidas directamente de la tabla de verdad, se pueden expresar mediante las siguientes ecuaciones:

$$s = \overline{x}\, y + x\, \overline{y} = x \oplus y$$

$$c = x\, y$$

donde el operador $\oplus$ representa la función OR-exclusiva.

El bit de suma s es 1 cuando x o y pero no ambos es 1, y el bit de arrastre c es 1 cuando ambos x e y son 1. El SSB se puede realizar de múltiples formas dependiendo de las puertas lógicas que se tengan, y de si se dispone o no del complemento de los bits de entrada. En la Figura 4.5 se muestran diferentes realizaciones del SSB.

4.1.2 Sumador binario completo (SBC)

Se diferencia del SSB porque tiene una tercera entrada c_{i-1} llamada "arrastre de la etapa anterior", que le permite encadenarse con otros SBC para el diseño de circuitos de suma de números de n bits ($n > 1$).

El SBC acepta como entradas un bit de cada uno de los operandos (x_i e y_i) y un bit de arrastre (c_{i-1}) de la etapa previa, y genera como salida un bit de suma (s_i) y un bit de arrastre (c_i) para la etapa siguiente de acuerdo con la Tabla 4.3. En la Figura 4.6 se da el diagrama de bloques y la representación de puntos de un SBC.

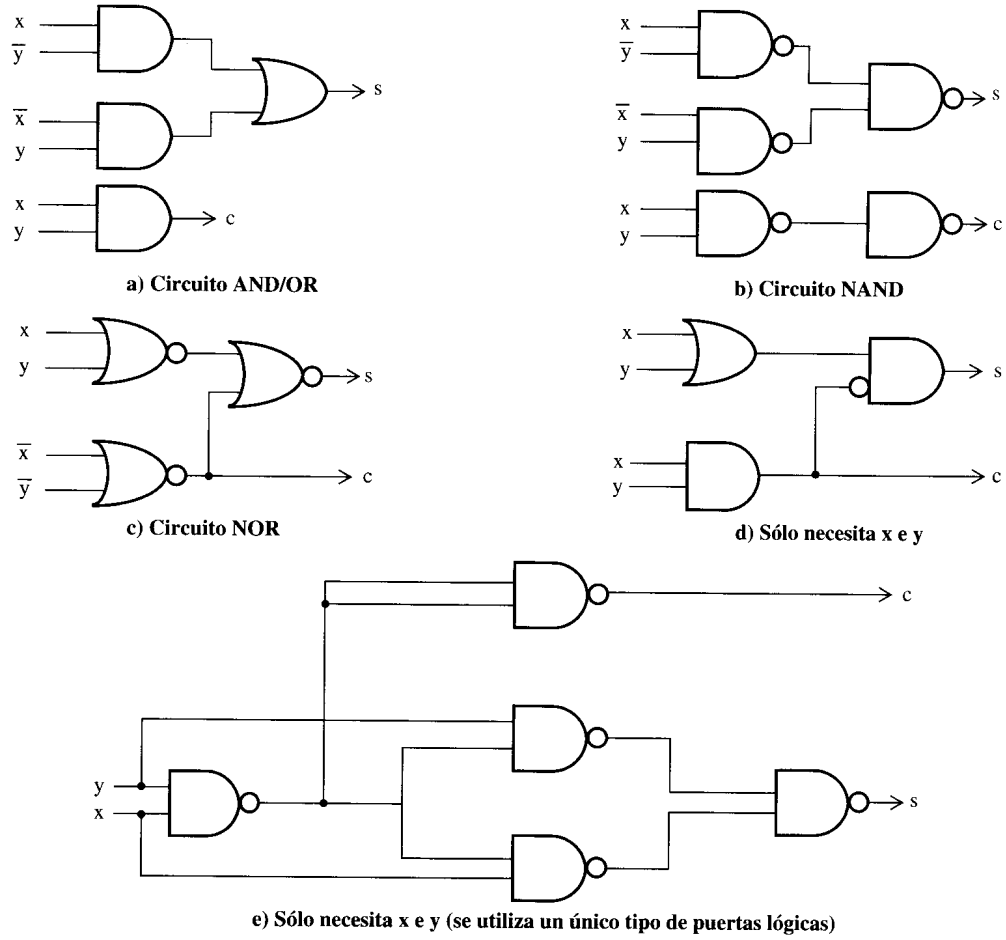


Figura 4.5: Diferentes realizaciones lógicas de un SSB

x_i	y_i	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabla 4.3: Tabla de verdad del SBC

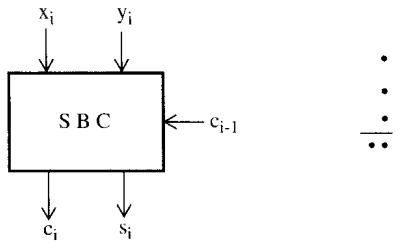


Figura 4.6: Diagrama de bloques y representación de puntos de un SBC

Las funciones lógicas de s_i y c_i, deducidas del mapa de Karnaugh de la Figura 4.7a , se pueden expresar mediante las siguientes ecuaciones:

$$s_i = \overline{x}_i\ \overline{y}_i\ c_{i-1} + \overline{x}_i\ y_i\ \overline{c}_{i-1} + x_i\ \overline{y}_i\ \overline{c}_{i-1} + x_i\ y_i\ c_{i-1}$$

$$c_i = x_i\ y_i + x_i\ c_{i-1} + y_i\ c_{i-1}$$

con lo que s_i y c_i se sintetizan mediante los circuitos lógicos de dos niveles que se muestran en la Figura 4.7b :

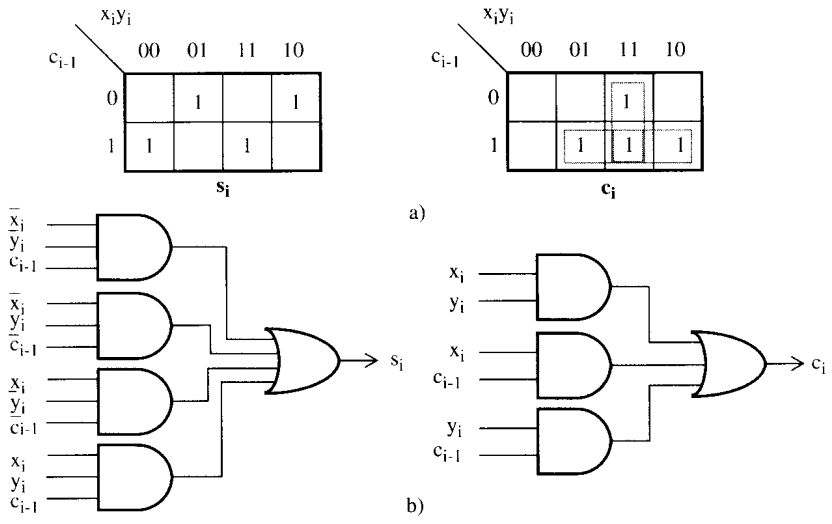


Figura 4.7: Diseño lógico de un SBC a) Mapa de Karnaugh b) Circuito lógico

El SBC también se puede realizar utilizando 2 SSB's conectados en cascada, tal como se muestra en la Figura 4.8, pero el circuito obtenido resulta más lento, debido a que las señales de entrada tienen que atravesar más niveles de puertas para generar la salida. A partir de las expresiones de s_i y c_i deducidas anteriormente se puede obtener:

$$s_i = (\overline{x}_i\ \overline{y}_i + x_i\ y_i)\ c_{i-1} + (x_i\ \overline{y}_i + \overline{x}_i\ y_i)\ \overline{c}_{i-1} = x_i \oplus y_i \oplus c_{i-1} \tag{4.1}$$

$$c_i = x_i\ y_i + (x_i\ \overline{y}_i + \overline{x}_i\ y_i)\ c_{i-1} = x_i\ y_i + (x_i \oplus y_i)\ c_{i-1} \tag{4.2}$$

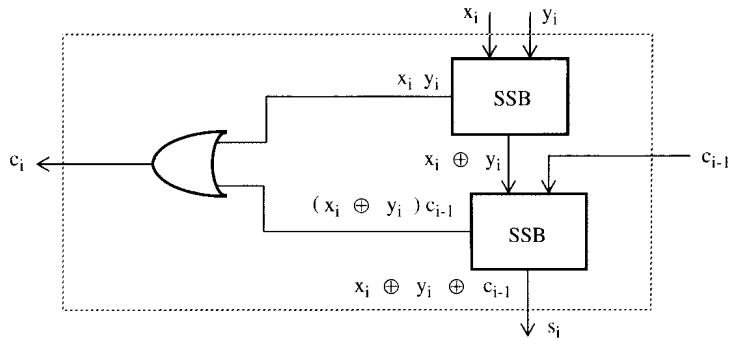


Figura 4.8: Realización de un SBC con 2 SSB´s

En la Tabla 4.4 se dan los retardos (medidos como el número de niveles de puertas lógicas que hay que pasar desde las entradas hasta las salidas) para los circuitos de las Figuras 4.7 y 4.8.

	s_i	c_i
Realización en 2 niveles	3	2
Realización con 2 SSB's	6	5

Tabla 4.4: Niveles de retardo en las dos realizaciones del SBC

4.1.3 Sumador binario serie

Con un SBC de 1 bit se puede diseñar un circuito sencillo que realiza la suma de dos números de n bits. Los bits del mismo peso, x_i e y_i, de los dos números a sumar se deben introducir sucesivamente en la entrada del SBC, comenzando por los bits menos significativos x_0 e y_0 (ver Figura 4.9).

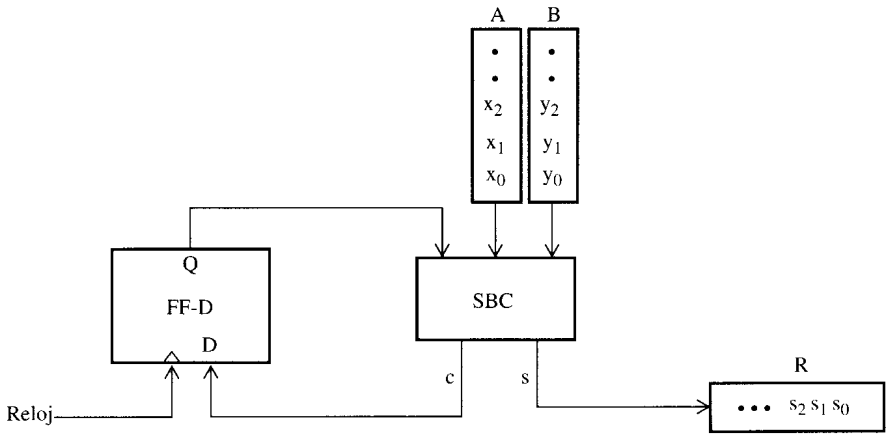


Figura 4.9: Sumador binario serie

Es necesario incluir un retardo unitario (elemento de memoria) para retener el arrastre c producido por el SBC de un intervalo al siguiente. Al comenzar la suma, la salida Q del elemento de memoria se inicializa a 0. El registro R donde se almacena el resultado puede ser el mismo registro A, en cuyo caso se denomina registro *acumulador*. Si se representa por Δ el retardo producido por el elemento de memoria (fijado fundamentalmente por la frecuencia del reloj), y por δ el tiempo que tarda el SBC en producir la suma de un bit, el tiempo que se necesita para realizar la suma de dos números de n bits será $n(\Delta + \delta)$. En un sumador binario serie la complejidad del circuito es independiente del número n de bits que hay que sumar, pero no sucede lo mismo con el tiempo de cálculo que crece linealmente con n.

4.1.4 Sumador binario paralelo con propagación del arrastre

La mayoría de las aplicaciones requieren sumas más rápidas que las que se pueden conseguir con los sumadores binarios serie. La manera clásica de resolver este problema es conectando una cadena de SBC's, de forma que se introduzcan en paralelo todos los bits de cada uno de los dos operandos.

Un sumador binario paralelo es un circuito que suma de forma simultánea todos los bits. Para sumar dos números de n bits, el circuito se puede realizar mediante el encadenamiento de n SBC's tal como se indica en la Figura 4.10.

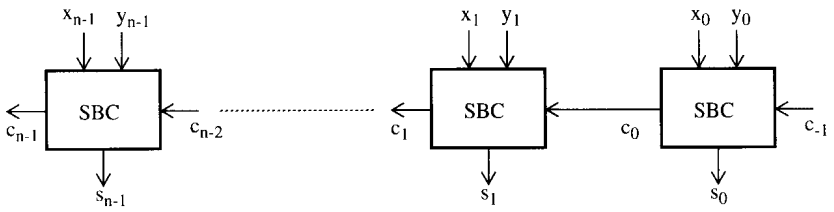


Figura 4.10: Sumador binario paralelo con propagación del arrastre (SPA)

Analizando la estructura de este sumador se observa que la salida s_{n-1} del SBC que opera sobre los bits más significativos no será válida hasta que se conozca el arrastre c_{n-2} producido por el SBC de la etapa anterior. Este mismo razonamiento se puede aplicar a cualquier etapa del sumador binario paralelo, por lo que en el caso más desfavorable, es decir, cuando se propaga un arrastre desde la etapa menos significativa hasta la más significativa, el resultado de la suma no será efectivo hasta que haya pasado un tiempo $\tau = n\delta$, siendo δ el tiempo que tarda un SBC en generar el arrastre para la etapa siguiente. En la Figura 4.11 se muestra un ejemplo del caso más desfavorable para $n = 8$.

```
  01001101
  10110011
-----------
 100000000
```

Figura 4.11: Ejemplo de caso más desfavorable de propagación del arrastre

Por este motivo, a este tipo de sumador binario paralelo se le conoce como *sumador con propagación del arrastre* (SPA). El retardo resultante en el sumador binario paralelo depende linealmente de n, por lo que será elevado para grandes valores de n y especialmente significativo cuando se conectan algunos módulos sumadores para formar un sumador más grande.

Se pueden conseguir sumadores binarios paralelos más rápidos que el sumador con propagación de arrastre utilizando circuitos que emplean más puertas lógicas con mayor número de entradas. La clave del compromiso rapidez-coste en los sumadores de alta velocidad estriba en el mecanismo utilizado para acelerar la generación de los arrastres.

4.1.5 Sumador-restador binario paralelo con propagación del arrastre

El sumador de la Figura 4-10 funciona correctamente en el caso de números sin signo y de números positivos, porque el 0 del bit de signo de un número positivo tiene el mismo efecto que un 0 a la izquierda en un número sin signo. La forma adecuada de sumar números negativos (que tienen un 1 en el bit de signo) depende del tipo de representación que se utilice (magnitud signo, complemento a 1ó complemento a 2).

Como $x - y = x + (-y)$, si se dispone de un circuito que calcula la suma de números negativos se puede efectuar la operación de restar. En el caso de la representación de números negativos en complemento a 2 la resta resulta especialmente sencilla, porque el valor negativo es muy simple de realizar. En efecto, si $y = y_{n-1}\ y_{n-2} \cdots y_1\ y_0$ es un número entero representado en complemento a 2, entonces $-y$ se define por:

$$-y = \overline{y}_{n-1}\,\overline{y}_{n-2} \cdots \overline{y}_1\,\overline{y}_0 + 1$$

así pues, para obtener $-y$ a partir de y, se efectuan las dos operaciones siguientes:

a) Sustituir todos los bits de y por su complemento (es decir, cambiar $0 \rightarrow 1$ y $1 \rightarrow 0$)

b) Sumar 1 al número resultante

Teniendo esto en cuenta, se puede diseñar fácilmente un único circuito sumador-restador (ver Figura 4.12).

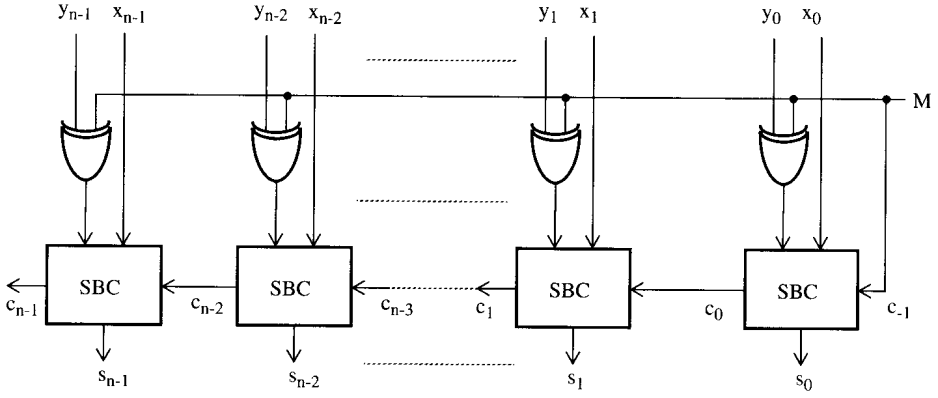


Figura 4.12: Sumador-restador binario paralelo con propagación de arrastre

Con cada módulo SBC se incluye una puerta OR-Exclusiva; la de la etapa i tiene como entradas M e y_i. La entrada M controla la operación. Si $M = 0$ el circuito es un sumador y cuando $M = 1$ el circuito se convierte en un restador. Cuando $M = 0$, como $y_i \oplus 0 = y_i$, los SBC's reciben el valor de y, la entrada de arrastre c_{-1} es 0, y el circuito realiza la operación $x + y$.

Cuando $M = 1$, como $y_i \oplus 1 = \overline{y}_i$ y $c_{-1} = 1$, se complementan todos los bits del número y se suma 1 debido

al arrastre de entrada del circuito. El circuito realiza la operación $x + C2(y)$, donde $C2(y)$ es el complemento a 2 de y. Para números sin signo, esto representa:

$$\begin{array}{lll} (x - y) & \text{si} & x \geq y \\ C2(y - x) & \text{si} & x < y \end{array}$$

Para números con signo el resultado es $(x - y)$ si no se produce rebose (overflow). La simplicidad del circuito lógico y la rapidez para efectuar tanto la suma como la resta de números con signo en complemento a 2, es la razón de por qué esta representación es la más utilizada en las unidades aritméticas de los computadores actuales.

Podría pensarse que la representación en complemento a 1 es tan buena como la de complemento a 2 para utilizarla en un circuito sumador-restador combinado. No obstante, aunque en este caso la complementación es trivial, el resultado obtenido después de la suma no es siempre correcto. No se puede ignorar el arrastre de salida c_{n-1}. Si $c_{n-1} = 1$, hay que sumar 1 al resultado para corregirlo. Si $c_{n-1} = 0$, el resultado que se obtiene es correcto. El requisito de este ciclo de corrección, que es condicional según sea el arrastre de salida de la suma, hace más complicado el circuito sumador-restador en complemento a 1 que en complemento a 2.

Detección del rebose en el circuito sumador-restador con propagación del arrastre

Cuando se suman números sin signo el arrastre de la última etapa sirve como un indicador de rebose. Sin embargo, en el caso de números con signo la cosa es algo más complicada. Obviamente, la suma de números con signos distintos no puede nunca originar un rebose, porque el valor aboluto de la suma será siempre más pequeño que el mayor valor absoluto de los dos sumandos. En el caso de que los sumandos tengan el mismo signo es cuando se puede producir un rebose; pueden ocurrir dos casos:

a) *x e y son positivos.* Como los bits de signo x_{n-1} e y_{n-1} son iguales a cero, no se puede generar ningún arrastre en la posición del bit de signo ($c_{n-1} = 0$). Sin embargo, el bit de signo de la suma s_{n-1} será 1 si se genera o propaga un arrastre desde la etapa anterior ($c_{n-2} = 1$) y el resultado es incorrecto (la suma de dos números positivos da un número negativo).

b) *x e y son negativos.* Como los bits de signo x_{n-1} e y_{n-1} son iguales a uno, se genera siempre un arrastre en la posición del bit de signo ($c_{n-1} = 1$). El bit de signo de la suma s_{n-1} será 0 si no se genera o no se propaga un arrastre desde la etapa anterior ($c_{n-2} = 0$) y el resultado es incorrecto (la suma de dos números negativos da un número positivo).

La Tabla 4.5 resume las condiciones de rebose expresadas en a) y b). La expresión lógica de la señal de rebose R deducida directamente de la Tabla 4-5 es:

$$R = x_{n-1}\, y_{n-1}\, \overline{s}_{n-1} + \overline{x}_{n-1}\, \overline{y}_{n-1}\, s_{n-1} \tag{4.3}$$

Es posible expresar la señal de rebose R como una función de los arrastres c_{n-2} y c_{n-1} mediante la siguiente función lógica:

$$R = c_{n-2} \oplus c_{n-1} \tag{4.4}$$

En la Figura 4.13 se muestra el circuito sumador-restador de la Figura 4-12 con el circuito de detección de rebose incorporado. Es importante detectar la aparición de un rebose en un computador, para poder decidir qué acción realizar cuando éste se produzca. Por esta razón es usual que la unidad aritmética disponga de un registro de 1 bit (registro de rebose) que se pone a 1 cuando se genera un rebose.

x_{n-1}	y_{n-1}	s_{n-1}	Rebose (R)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Tabla 4.5: Condiciones de rebose

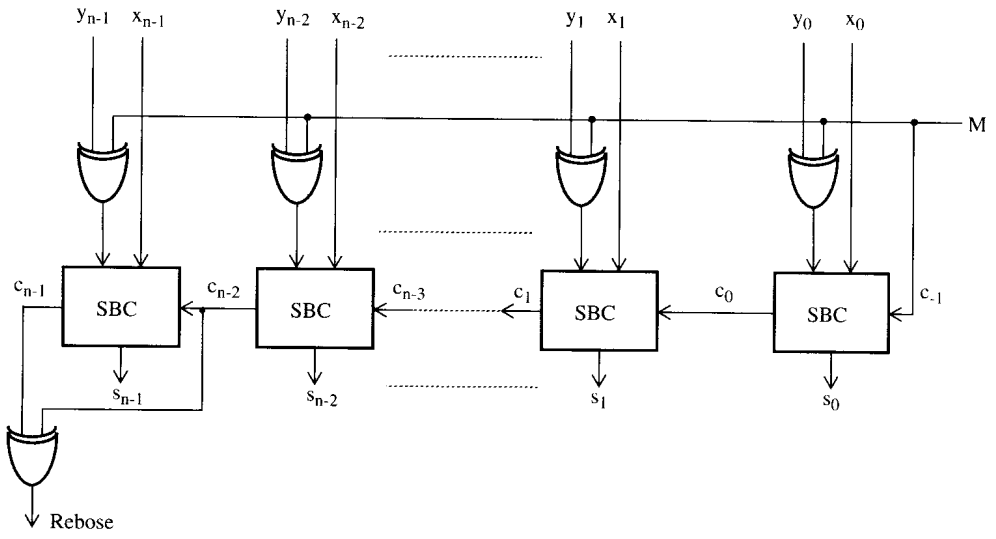


Figura 4.13: Sumador-restador binario paralelo con propagación del arrastre y detección del rebose

Sección 4-2

Sumadores de alta velocidad

La naturaleza secuencial de la propagación del arrastre es el problema más difícil que hay que resolver cuando se desea acelerar la suma. Las diferentes estrategias que se han propuesto para resolver este problema se pueden clasificar de la forma siguiente:

1) *Aceptación de la existencia de los arrastres.* En lugar de intentar acelerar la generación de los arrastres simplemente se les acepta tal como son. Se diseña el sumador de manera que su salida no se utiliza hasta que haya pasado el tiempo suficiente para que se generen los arrastres cuando se produce su secuencia más desfavorable (es decir que los arrastres se hayan propagado desde el bit menos significativo al más significativo de los números a sumar). En algunos microprocesadores de 8 bits su frecuencia de reloj se ajusta de forma tal, que las salidas del sumador no se utilizan hasta que la secuencia más larga de posibles arrastres ha tenido tiempo de propagarse, esté o no realmente presente.

2) *Anticipación del arrastre.* Los sumadores de esta clase poseen una circuitería extra que "desvía" la generación de los arrastres de las etapas del sumador paralelo (o grupo de etapas) que lo que hacen es propagarlos.

3) *Suma condicional.* Análogamente a los sumadores con anticipación de arrastre, este tipo de sumadores produce la suma en un tiempo $O(\log_2 n)$, siendo n la longitud de los operandos. En pasos sucesivos se obtienen 1, 2, 4, ..., 2^k dígitos correctos hasta que se completa la suma.

4) *Detección de la finalización del arrastre.* Añadiendo un circuito lógico adicional es posible detectar cuando ha finalizado la propagación de los arrastres a lo largo del sumador. En ese momento las salidas del sumador tendrán su valor final correcto y se podrán utilizar.

5) *Minimización del número de arrastres.* Cuanto mayor sea la base del sistema de numeración utilizado para representar los operandos, menor será el número de arrastres que se producen durante la suma. Por este motivo se pueden considerar sistemas de numeración de base 2^k, para los que k dígitos binarios consecutivos corresponden a un dígito en dicha base. La operación de suma se efectúa directamente en la nueva base.

6) *Arrastre almacenado.* Cuando el número de sumandos es mayor que 2 es posible retardar la propagación del arrastre hasta el final de la operación. De esta manera, se puede considerar que el tiempo necesitado para la asimilación de los arrastres diferidos se reparte uniformemente entre todos los operandos.

4.2.1 Características de los arrastres

Antes de considerar algunos de los métodos mencionados en el apartado anterior para acelerar la suma, conviene examinar con cierto detalle las características de los arrastres.

a) Un arrastre se *generará* en la posición i-ésima si se cumple que la suma aritmética de los bits correspondientes (x_i e y_i) es mayor que uno, es decir, si $x_i + y_i > 1$.

b) Un arrastre que llega a la posición i-ésima desde la posición (i-1)-ésima, se *propagará* a la posición (i+1)-ésima si se cumple que la suma aritmética de los bits correspondientes (x_i e y_i) es igual a uno, es decir, si $x_i + y_i = 1$.

	$y_i = 0$	$y_i = 1$
$x_i = 0$	-	p
$x_i = 1$	p	g

Tabla 4.6: Tipos de arrastres

La Tabla 4.6 muestra de forma resumida las condiciones bajo las cuales no hay arrastres (-), se propaga el arrastre recibido (p) o se genera un arrastre (g). Estas tres acciones (-, p y g) son mutuamente excluyentes, de manera que solamente puede ocurrir una de ellas en cada posición. De la Tabla 4.6 se sigue que aquellas etapas del sumador para las que $x_i = y_i = 1$ comenzarán una secuencia de arrastre. La propagación de los arrastres continuará a través de las etapas en las que $x_i \neq y_i$ y parará cuando llegue a una etapa en la que $x_i = y_i$. El ejemplo de la Figura 4.14 muestra 4 secuencias de arrastre de longitudes 2, 4, 1 y 2 que comienzan todas simultáneamente.

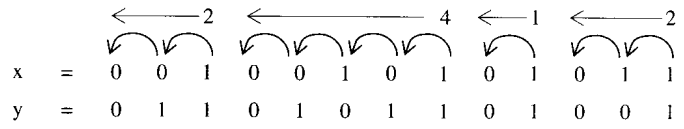


Figura 4.14: Secuencias de arrastres

Esta última consideración, de la concurrencia de la secuencia de arrastres, es importante y se usa en algunas estrategias para acelerar la suma (sumadores con detección del final de los arrastres). La secuencia de arrastre de longitud 2 a la izquierda de la Figura 4.14 y su adyacente de longitud 4 no es una única secuencia de longitud 6 sino dos secuencias simultáneas.

4.2.2 Sumadores con anticipación del arrastre

Es la forma más extendida de diseñar sumadores de alta velocidad. Su principio básico consiste en reducir el retardo producido por la propagación de los arrastres de los SBC's de menor peso a los de mayor peso. La forma de hacerlo es generando la entrada de arrastre de la etapa i-ésima directamente a partir de los bits de entrada a las etapas precedentes i - 1, i - 2, ..., 1, 0 en lugar de tener que esperar a que el arrastre se propague a lo largo de dichas etapas (de ahí el nombre de anticipación del arrastre).

En la ecuación (4.2) de la salida c_i del arrastre de un SBC cuando se realiza a partir de dos SSB's, se pueden distinguir dos términos. El primero de ellos $x_i y_i$ depende solamente de las entradas de la etapa i y corresponde al arrastre generado en dicha etapa (g_i). El segundo de los términos establece que un arrastre procedente de la etapa previa (c_{i-1}) se propagará a la siguiente si $x_i \oplus y_i = 1$ y representa[2] el arrastre propagado por la etapa i (p_i). Por tanto la suma s_i y el arrastre c_i de la etapa i se pueden expresar por las expresiones:

$$s_i = p_i \oplus c_{i-1} \qquad c_i = g_i + p_i c_{i-1} \tag{4.5}$$

Si se observa la realización del SBC a partir de dos SSB's (ver Figura 4.8) se comprueba que el primer SSB genera los términos g_i y p_i, por lo que se puede diseñar un SBC modificado que añade a sus salidas estos términos tal como se muestra en la Figura 4.15.

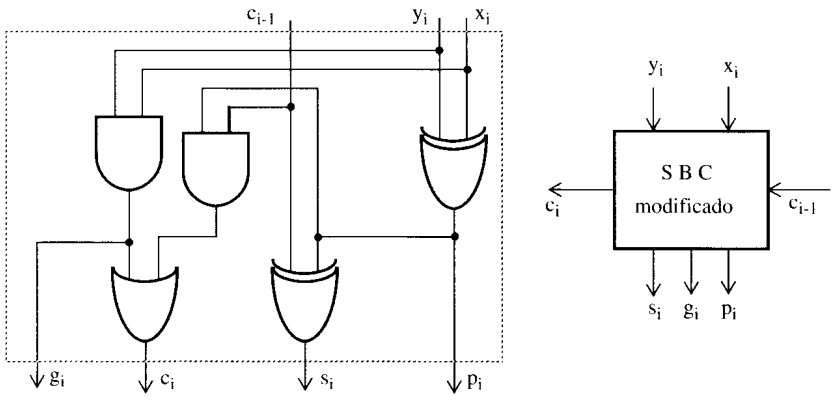


Figura 4.15: SBC modificado

De la ecuación (4.5) se deduce que todos los bits de la suma se generan en paralelo si todas las entradas de arrastre c_{n-2}, ..., c_0, c_{-1} están disponibles de forma simultánea. Si se aplica la ecuación (4.5) recursivamente se tiene:

$$c_0 = g_0 + p_0 c_{-1}$$

$$c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{-1}$$

$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{-1}$$

.....

$$c_i = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_1 g_0 + p_i p_{i-1} \ldots p_0 c_{-1}$$

como puede apreciarse, cada término producto que aparece en c_i es la conjunción de generación y propagaciones de arrastres. Así, en la etapa i, se produce un arrastre c_i si:

1) La etapa i lo genera ($g_i = 1$)

2. En algunos textos se utiliza en lugar de la función de propagación del arrastre $p_i = x_i \oplus y_i$ la función de *transferencia del arrastre* $t_i = x_i + y_i$.

2) La etapa i - k (k = 1, 2,..., i) lo genera (g_{i-k} = 1) y las etapas desde i - k + 1 hasta i, lo propagan (p_{i-k+1} = 1, ..., p_{i-1} = 1, p_i = 1)

3) Hay un arrastre inicial (c_{-1} = 1) y todas las etapas lo propagan (p_0 = 1, ..., p_{i-1} = 1, p_i = 1)

En la Figura 4.16 se muestra un circuito combinacional que realiza las ecuaciones anteriores para el caso de 4 bits. A este circuito se le denomina *circuito de aceleración de arrastres* (CAA). El retardo para la obtención de los arrastres en este circuito es de sólo dos niveles de puertas lógicas.

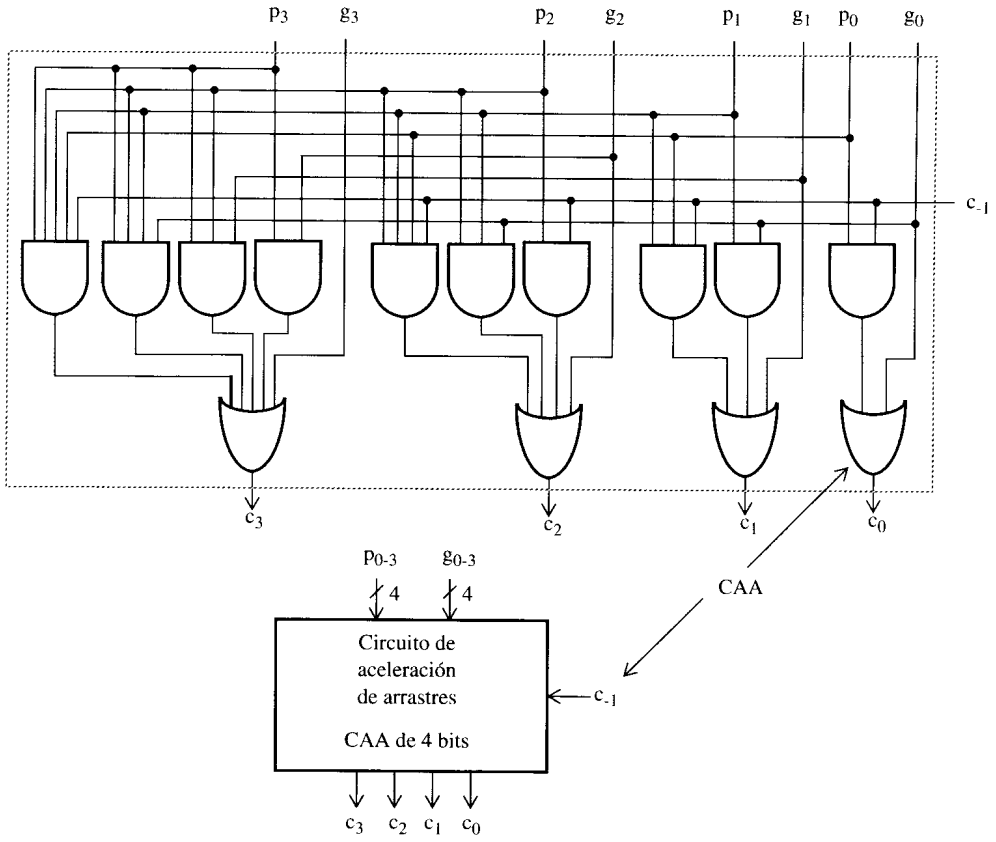


Figura 4.16: Circuito de aceleración de arrastres (CAA) de 4 bits

Si se combinan 4 módulos del tipo SBC modificado de la Figura 4.15 y el CAA de la Figura 4.16 se obtiene un *sumador con aceleración de los arrastres* (SAA) de 4 bits, tal como se muestra en la Figura 4.17. Se puede establecer la siguiente relación:

$$1 \text{ SAA de } n \text{ bits} = \{n \text{ SBC's modificados}\} + 1 \text{ CAA de } n \text{ bits}$$

A medida que aumenta el número de bits del sumador va creciendo la complejidad del módulo CAA, por lo que en la práctica, para sumadores con un número elevado de bits no resulta viable la aplicación estricta de esta técnica. Una solución a este problema consiste en encadenar m SAA's de 4 bits con propagación de los arrastres (como el de la Figura 4.17) para formar sumadores de $n = 4 \times m$ bits. En la Figura 4.18 se muestra un sumador de 16 bits construido con esta técnica.

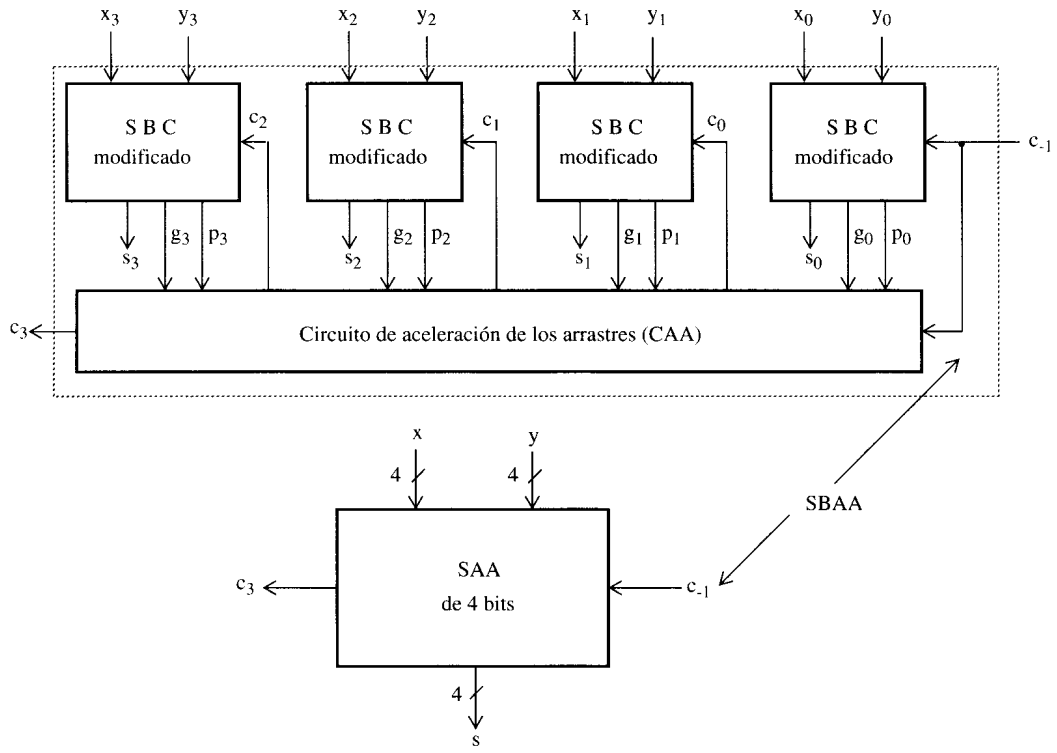


Figura 4.17: Sumador con aceleración de los arrastres de 4 bits (SAA)

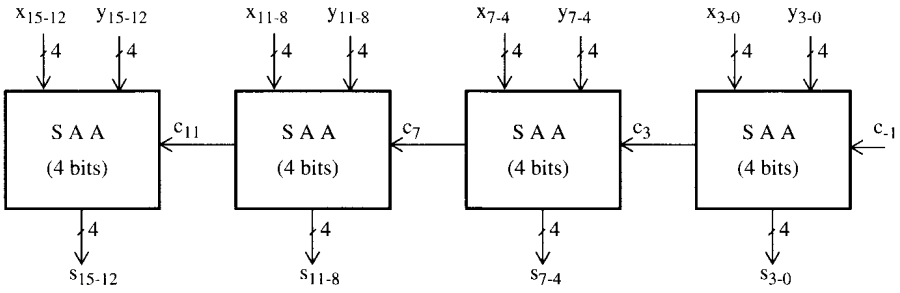


Figura 4.18: Sumador de 16 bits construido con 4 SAA's de 4 bits

Se observa que el SAA de mayor peso no puede dar su resultado mientras no conozca c_{11}, pero siempre el retardo que se incurre en la propagación de los arrastres es menor que si se dispone de 16 SBC's en cascada. A este tipo de sumadores se les conoce como *sumadores con propagación anticipada del arrastre.*

4.2.3 Sumadores de suma condicional

El principio en el que se fundamenta el sumador de suma condicional es generar dos conjuntos de salidas. Cada conjunto incluye k bits de suma y un arrastre de salida. Uno de los conjuntos asume que eventualmente el arrastre de entrada será cero, mientras que el otro supone que será uno. Una vez que se conozca el arrastre de entrada, se necesita seleccionar solamente el conjunto correcto de salidas (uno de los dos conjuntos calculados) sin tener que esperar a que el arrastre se propague a lo largo de las k posiciones. Obviamente no se debería aplicar esta idea a todos los n bits de los operandos al comienzo de la operación de suma ya que entonces se tendría que esperar hasta que el arrastre se propague a través de los n bits antes de que se pueda hacer la selección. Se necesita por lo tanto dividir los n bits dados en grupos más pequeños y aplicar la idea anterior a cada uno de ellos por separado. De esta forma, se puede hacer en paralelo la propagación del arrastre dentro de cada uno de los grupos lo que reduce el tiempo total de ejecución. Estos grupos se pueden subdividir aún más en subgrupos, para los que el tiempo de propagación del arrastre es incluso más pequeño. Las salidas de los subgrupos se combinan entonces para generar la salida de los grupos.

Una división natural de los n bits de los operandos sería en dos grupos de $n/2$ bits. Cada uno de estos grupos se puede subdividir en dos subgrupos de $n/4$ bits. Este proceso puede, en principio, continuarse hasta que se alcanza un grupo de tamaño 1 si n es una potencia de 2. En este caso se necesitan $k = \log_2 n$ pasos en el proceso, donde en el paso 1 se consideran los bits de los operandos de forma individual, en el paso 2 se tratan parejas de bits y así sucesivamente. El sumador de suma condicional proporciona así el resultado correcto de la suma en un tiempo $O(\log_2 n)$. Conviene observar, sin embargo, que un grupo dado no tiene necesariamente que dividirse en subgrupos de igual tamaño. Así pues el esquema de suma condicional se puede aplicar incluso si el número de bits no es una potencia de 2. Se ilustra a continuación la forma en que los grupos de 1 sólo bit se combinan en parejas de bits. Se utiliza la siguiente notación:

s_i^0 = bit de suma en la posición i, si el arrastre de entrada al grupo es 0

s_i^1 = bit de suma en la posición i, si el arrastre de entrada al grupo es 1

c_{i+1}^0 = arrastre de salida del grupo, si el arrastre de entrada al grupo es 0

c_{i+1}^1 = arrastre de salida del grupo, si el arrastre de entrada al grupo es 1

En la Tabla 4.7 se consideran dos posiciones de bits adyacentes. En el paso 1 cada una de las posiciones constituye un grupo separado.

i	7	6	
x_i	1	0	
y_i	0	0	
s_i^0	1	0	Suponiendo que el arrastre de entrada = 0
c_{i+1}^0	0	0	
s_i^1	0	1	Suponiendo que el arrastre de entrada = 1
c_{i+1}^1	1	0	

Tabla 4.7: Paso 1 del algoritmo de suma condicional para dos posiciones de bits adyacentes

En el paso 2, los dos bits se combinan en un grupo de tamaño 2 (ver Tabla 4.8)

i, i - 1	7, 6	
x_i , x_{i-1} y_i , y_{i-1}	1 0 0 0	
s_i^0 , s_{i-1}^0 c_{i+1}^0	1 0 0	Suponiendo que el arrastre de entrada es 0
s_i^1 , s_{i-1}^1 c_{i+1}^1	1 1 0	Suponiendo que el arrastre de entrada es 1

Tabla 4.8: Paso 2 del algoritmo de suma condicional para una pareja de bits

En la Tabla 4.9 se muestra un ejemplo de suma condicional de dos números de 8 bits. El proceso tiene $\log_2 8 = 3$ pasos. El arrastre forzado (que es igual a 0 en este ejemplo) está disponible al comienzo de la operación, por lo que sólo se necesita que se genere un conjunto de salidas para el grupo de más a la derecha en cada paso.

	i	7	6	5	4	3	2	1	0
	x_i y_i	1 0	0 0	1 1	1 0	0 1	1 1	1 0	0 1
Paso 1	s_i^0 c_{i+1}^0	1 0	0 0	0 1	1 0	1 0	0 1	1 0	1 0
	s_i^1 c_{i+1}^1	0 1	1 0	1 1	0 1	0 1	1 1	0 1	
Paso 2	s_i^0 c_{i+1}^0	1 0	0	0 1	1	0 1	0	1 0	1
	s_i^1 c_{i+1}^1	1 0	1	1 1	0	0 1	1		
Paso 3	s_i^0 c_{i+1}^0	1 0	1	0	1	0 1	0	1	1
	s_i^1 c_{i+1}^1	1 0	1	1	0				
Resultado		1	1	1	0	0	0	1	1

Tabla 4.9: Suma condicional de dos números de 8 bits

4.2.4 Sumadores con selección del arrastre

El sumador con selección del arrastre se basa esencialmente en el mismo principio que el sumador de suma condicional, es decir en la generación de más de un conjunto de bits de suma seguida por la selección de uno de ellos de acuerdo con los arrastres determinados. En realidad, el sumador de suma condicional se puede considerar como un caso especial del sumador con selección del arrastre. En el primero la formación inicial de los bits de suma ocurre con grupos de 1 bit y la selección de los bits de suma que son correctos es a través de un árbol en el que el tamaño del grupo se duplica en cada paso. En el segundo la formación inicial de los bits de suma ocurre en grupos de m bits ($m > 1$) y el tamaño del grupo utilizado durante la selección del arrastre puede aumentar por un factor mayor que dos.

En la Figura 4.19 se muestra un sencillo ejemplo de suma de dos números de 8 bits que ilustra el funcionamiento de los sumadores con selección del arrastre. El sumador de los 4 bits menos significativos puede ser un SPA (sumador con propagación del arrastre) o un SAA (sumador con anticipación del arrastre). Simultáneamente los 4 bits más significativos se suman una vez suponiendo que el arrastre de entrada es 1 y otra que es 0. Cuando se conoce el arrastre de salida de los 4 bits menos significativos, se puede utilizar para seleccionar mediante un multiplicador cuál de las dos secuencias calculadas es la correcta.

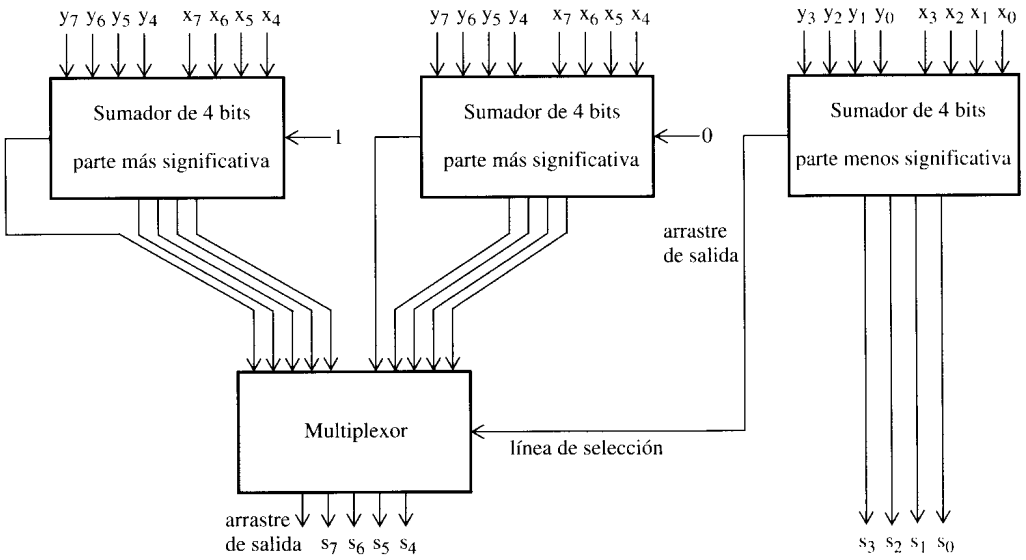


Figura 4.19: Diagrama de bloques de un sumador con selección del arrastre de 8 bits

4.2.5 Sumadores con detección de la finalización del arrastre

El sumador paralelo con *propagación del arrastre* se basa en una concepción síncrona (intervalo de suma constante para cualquier configuración de los operandos) que tiene en cuenta el caso más desfavorable de la propagación del arrastre. Esto implica que el tiempo de operación de la suma depende de forma lineal de n (número de bits de los sumandos).

Como un arrastre de máxima longitud (es decir, que se genera en la primera etapa y se propaga por todas las demás) no se da frecuentemente, en la mayoría de los casos sucederá que se espera de forma inútil un tiempo excesivo para el cálculo de la suma.

Ya en 1946 Burks, Goldstine y von Neumann, en su trabajo clásico sobre computadores con programa almacenado, analizaron el problema de la propagación del arrastre (usando números de 40 bits generados aleatoriamente), y demostraron que el "valor medio de la longitud máxima de propagación del arrastre" en la suma de dos números binarios de n bits es del orden de $\log_2 n$. Para $n = 32$ el valor medio será ≤ 5, mientras que para $n = 64$ será ≤ 6. La relación entre las longitudes de propagación del arrastre para el caso más desfavorable y el valor medio de la longitud máxima viene expresada por:

$$\tau = \frac{n}{\log_2 n}$$

Para $n = 16$ bits el retardo en un sumador con propagación del arrastre es de 32 (2 por etapa), mientras que en un sumador con detección de fin de arrastre el retardo es en promedio de 8 ($\log_2 16 = 4$ etapas $\times$ 2 retardo/etapa). Esto supone que en media la realización de la suma es cuatro veces más rápida.

El diseño de esta clase de sumador se basa en incorporar dentro del mismo, una unidad que detecte cuándo ha terminado la operación, y son por lo tanto de naturaleza asíncrona (ya que no es necesario un intervalo fijo para el tiempo de suma).

Además de los SBC's hay que añadir en cada etapa un circuito lógico que detecta los finales de arrastre. Cuando los sumadores individuales hayan acabado producirán un arrastre a la siguiente etapa (c_i) o un no arrastre (nc_i). La aparición de un c_i o un nc_i indica que esa etapa ha funcionado y cuando sucede en todas, señala la terminación de la operación. Al empezar la suma no existen ninguno de los dos arrastres (c_i o nc_i). En la Tabla 4.10 se dan las configuraciones que provocan la generación de c_i o nc_i y las de propagación de c_i.

x_i	y_i	c_{i-1}	c_i	nc_i	Arrastre
0	0	0	0	1	No se genera
0	0	1	0	1	$g_{0i} = \overline{x}_i \overline{y}_i$
0	1	0	0	0	
0	1	1	1	1	Se propaga
1	0	0	0	0	$p_i = x_i \oplus y_i$
1	0	1	1	1	
1	1	0	1	0	Se genera
1	1	1	1	0	$g_{1i} = x_i y_i$

Tabla 4.10: Generación de c_i y nc_i

Para controlar el comienzo de la suma se añade una señal I de inhibición. Tanto c_i como nc_i se inicializan a cero haciendo $I = 0$. Cuando $I = 1$ empieza la suma. Las expresiones lógicas de las señales c_i y nc_i son:

$$c_i = (x_i \oplus y_i)\, c_{i-1} + x_i\, y_i\, I = g_{1i}\, I + p_i\, c_{i-1} \tag{4.6}$$

$$nc_i = (x_i \oplus y_i)\, nc_{i-1} + \overline{x}_i\, \overline{y}_i\, I = g_{0i}\, I + p_i\, nc_{i-1} \forall\, i \neq 0 \tag{4.7}$$

y para $i = 0$:

$$c_0 = (x_0 \oplus y_0)\,(c_{-1}\,I) + x_0\,y_0\,I \tag{4.8}$$

$$nc_0 = (x_0 \oplus y_0)\,(nc_{-1}\,I) + \overline{x}_0\,\overline{y}_0\,I \tag{4.9}$$

el primer término de c_0 y nc_0 incluyen I, para garantizar que antes de empezar, cuando $I = 0$, se cumpla que $c_i = nc_i = 0\ \forall\ i$. En la Figura 4.20 se muestra el módulo de propagación del arrastre (PA) que realiza la síntesis de las ecuaciones (4.6) y (4.7).

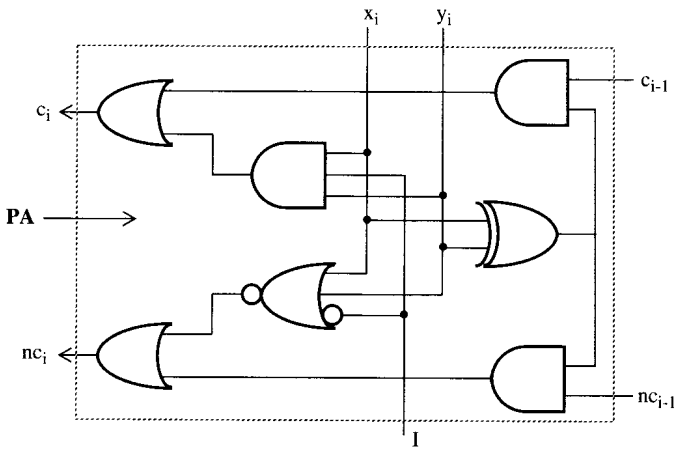


Figura 4.20: Celda de propagación del arrastre (PA)

Con el módulo PA se puede diseñar un sumador de n bits que detecta el fin del arrastre (ver Figura 4.21). La señal *FS* indica el fin de la suma porque en todos los módulos c_i ó nc_i son unos.

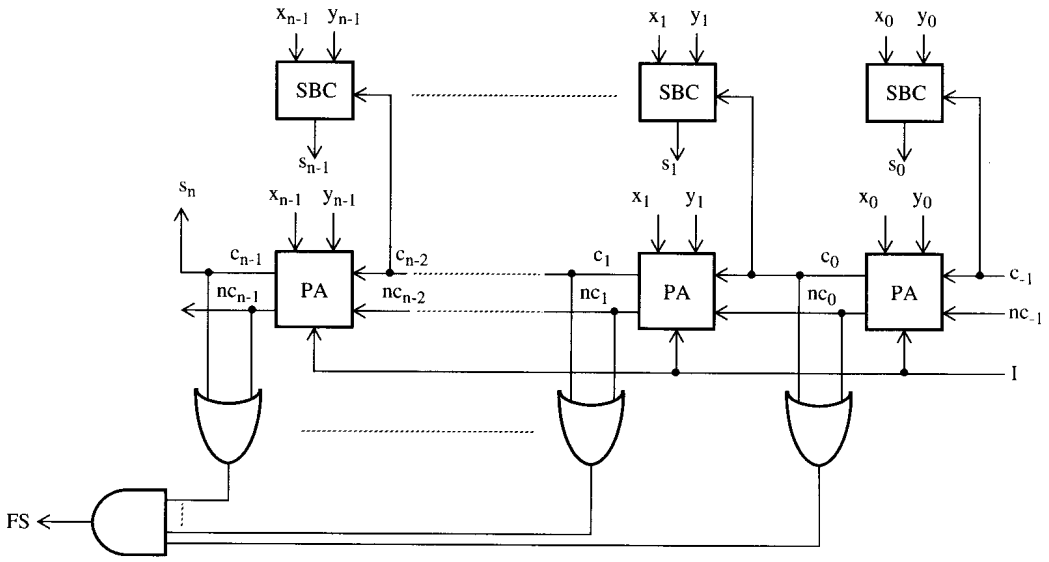


Figura 4.21: Sumador de n-bits con detector de fin de suma FS

Se genera un primer arrastre en la etapa i si $x_i = y_i = 1$, y se propaga hacia la izquierda pasando a través de las etapas donde $x_i\ y_i$ = 10 ó 01. Para evitar indicaciones erróneas deben darse simultáneamente las dos entradas al sumador. Cuando ha empezado la operación no pueden admitirse cambios en las entradas.

4.2.6 Sumadores que minimizan el número de arrastres

Cuando se suman dos números de 64 bits en un SBAA, la secuencia de propagación de arrastres más desfavorable es a través de las 64 etapas de que consta el sumador. Si los números se expresan en base 4, en lugar de hacerlo en base 2, tienen una longitud de 32 dígitos, por lo que la suma aritmética efectuada en dicha base también reduce la secuencia de arrastre más desfavorable a una longitud de 32.

En general, si se representan los números en base 2^p tendrán una longitud de $64/p$ dígitos, y si la suma aritmética se realiza en esa misma base la longitud de la secuencia de arrastre más desfavorable será de $64/p$. A pesar de la reducción que se produce en el tamaño de la secuencia de propagación de los arrastres a lo largo del sumador, cuando se opera en sistemas de numeración de base elevada, la circuitería lógica que se necesita para efectuar la suma aritmética en estas bases, es mucho más compleja que la empleada cuando se utiliza base 2. Además, es difícil conseguir circuitos aritméticos tan rápidos como los binarios.

Por ejemplo, un sumador que acepta como entrada dos grupos de 4 bits (cada grupo representa un dígito en base 16), tiene 9 entradas (incluyendo un arrastre de entrada) y 5 salidas (4 de la suma y 1 del arrastre de salida). No resulta viable realizar este circuito en dos niveles de puertas AND-OR ya que desde un punto de vista práctico no se construyen este tipo de puertas con más de 6 entradas.

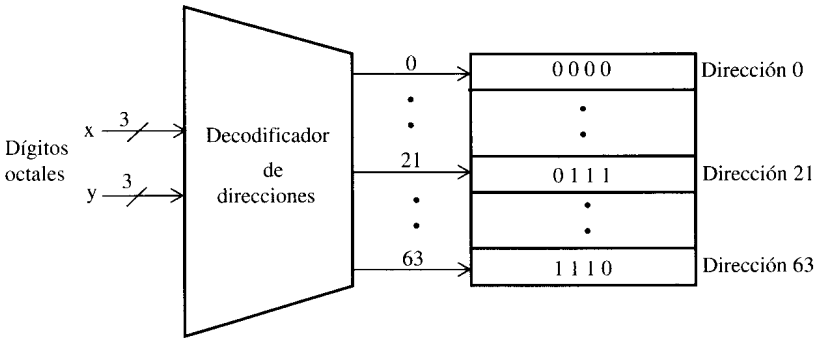
Una forma más atractiva de realizar la suma aritmética en bases superiores a 2 es utilizando una *tabla de consulta*, esto es, se proporciona una tabla que almacena el valor de la suma y de los arrastres para todos los posibles pares de sumandos. Si los operandos de entrada son números de m bits, dicha tabla tendría 2^m filas × 2^m columnas y 2^{2m} entradas con $(m+1)$ bits/entrada (ver Tabla 4.11 para el caso de una base octal, $m = 3$).

	$0_8 = 000_2$	$1_8 = 001_2$	$2_8 = 010_2$	$3_8 = 011_2$	$4_8 = 100_2$	$5_8 = 101_2$	$6_8 = 110_2$	$7_8 = 111_2$
$0_8 = 000_2$	$000\ 000_2$	$000\ 001_2$	$000\ 010_2$	$000\ 011_2$	$000\ 100_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$
$1_8 = 001_2$	$000\ 001_2$	$000\ 010_2$	$000\ 011_2$	$000\ 100_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$
$2_8 = 010_2$	$000\ 010_2$	$000\ 011_2$	$000\ 100_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$
$3_8 = 011_2$	$000\ 011_2$	$000\ 100_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$	$001\ 010_2$
$4_8 = 100_2$	$000\ 100_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$	$001\ 010_2$	$001\ 011_2$
$5_8 = 101_2$	$000\ 101_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$	$001\ 010_2$	$001\ 011_2$	$001\ 100_2$
$6_8 = 110_2$	$000\ 110_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$	$001\ 010_2$	$001\ 011_2$	$001\ 100_2$	$001\ 101_2$
$7_8 = 111_2$	$000\ 111_2$	$001\ 000_2$	$001\ 001_2$	$001\ 010_2$	$001\ 011_2$	$001\ 100_2$	$001\ 101_2$	$001\ 110_2$

Tabla 4.11: Suma de 2 números en base octal

Se necesita pues una memoria ROM de 2^{2m} palabras × $(m + 1)$ bits (de la Tabla 4.11 no hay que almacenar los dos bits más significativos ya que son siempre iguales a 0). Los 2 sumandos concatenados dan la dirección de memoria cuyo contenido es precisamente su suma (ver Figura 4.22 para el ejemplo anterior).

Esta técnica de emplear una ROM como tabla de consulta en la síntesis de circuitos combinacionales, encuentra múltiples aplicaciones en otros módulos aritméticos tales como los sumadores con múltiples operandos y los multiplicadores binarios que se verán posteriormente.



Ejemplo $x = 0\ 1\ 0$, $y = 1\ 0\ 1$ $\Rightarrow$ $x + y = 0\ 1\ 1\ 1$ es el contenido de la dirección $xy = 0\ 1\ 0\ 1\ 0\ 1_2 = 21_{10}$

Figura 4.22: ROM de 64 palabras × 4 bits/palabra para sumar 2 dígitos octales

4.2.7 Sumadores con arrastre almacenado

La suma de dos números binarios de n bits x^0 e y^0 se puede ver como un proceso iterativo en el que en cada paso i las salidas que se producen x^i e y^i, $i = 1, 2, \ldots$. viene dada por:

$$x^i(j) = x^{i-1}(j) \oplus y^{i-1}(j - 1) \qquad j = 0, 1, .., n - 1$$

$$y^i(j) = x^{i-1}(j) \text{ AND } y^{i-1}(j - 1) \qquad j = 0, 1, .., n - 1$$

donde $x^i(j)$ e $y^i(j)$ indican el bit j de las salidas en la etapa i. Las dos ecuaciones anteriores representan la función de un SSB que recibe como entrada el bit de suma del paso previo ($x^{i-1}(j)$) y el bit de arrastre del paso previo desplazado un lugar a la izquierda ($y^{i-1}(j\text{-}1)$). A continuación, en un ejemplo con dos numeros de 5 bits se muestra paso a paso el proceso iterativo de la suma de acuerdo con las expresiones anteriores.

0 1 1 0 1	1er sumando x^0
0 0 1 0 1	2° sumando y^0
0 1 0 0 0	1ª suma parcial x^1
0 0 1 0 1	1er arrastre parcial y^1
0 1 0 1 0	arrastre parcial desplazado
0 0 0 1 0	2ª suma parcial x^2
0 1 0 0 0	2° arrastre parcial y^2
1 0 0 0 0	arrastre parcial desplazado
1 0 0 1 0	3ª suma parcial x^3
0 0 0 0 0	3er arrastre parcial y^3 (todos ceros)

En este punto finaliza la suma porque el arrastre parcial es 0 en todas las posiciones. El procedimiento se puede mecanizar utilizando de forma repetida un grupo de SSB's o mediante una batería de sucesivos grupos de SSB's. En la Figura 4.23 se muestra un diagrama de bloques de esta solución. Inicialmente los 2 registros R_a y R_b contienen los números x e y. En los pasos siguientes, los contenidos de estos registros se sustituyen por la suma parcial x^j y el arrastre parcial y^j desplazado un lugar hacia la izquierda.

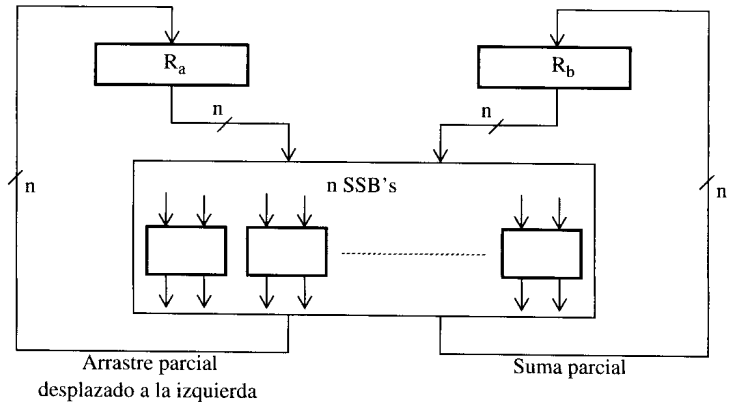


Figura 4.23: Suma de 2 números con arrastre almacenado

Este circuito no resulta útil, pues aunque emplea solamente n SSB's, es más lento que si se emplean n SBC's conectados directamente (ver Figura 4.10) debido al tiempo para las transferencias a los registros.

Cuando se tienen que sumar tres o más operandos simultáneamente (por ejemplo en la multiplicación) utilizando sumadores con sólo dos operandos, el tiempo que se emplea en la propagación del arrastre debe repetirse algunas veces. Si el número de operandos es k, entonces los arrastres tienen que propagarse (k - 1) veces. Se han propuesto e implementado diferentes técnicas para la suma de operandos múltiples que intentan reducir el retardo que impone la propagación de los arrastres. Una de las más utilizadas es la de los *sumadores con arrastre almacenado* o *sumadores CSA* (Carry Save Adder). En esta clase de sumadores el arrastre sólo se propaga en el último paso del proceso de suma, mientras que en los pasos anteriores se genera de forma separada una suma parcial y una secuencia de arrastres. Así, un CSA toma como entrada tres operandos de n bits y genera como resultado dos secuencias de n bits cada una, la primera corresponde a la suma parcial y la segunda a la secuencia de los arrastres. Un segundo CSA acepta estas dos secuencias anteriores y otro operando de entrada y produce como salida una nueva suma parcial y una nueva secuencia de los arrastres. Un CSA es por lo tanto capaz de reducir el número de operandos que se suman de 3 a 2 sin utilizar ninguna propagación de los arrastres.

Un CSA se puede implementar de diferentes maneras. En su forma más simple, el elemento básico de un CSA es el SBC (sumador binario completo) con tres entradas x, y y z. La operación aritmética que realiza se puede describir como:

$$x + y + z = 2c + s$$

donde s y c son la suma y el arrastre respectivamente. Sus valores son:

$$s = (x + y + z) \bmod 2 \qquad \text{y} \qquad c = \frac{(x + y + z) - s}{2}$$

Las salidas son la representación binaria del número de 1´s que hay en las entradas. De esta forma se puede considerar, tal como se muestra en la Figura 4.24, que un SBC es un contador C(3,2). El primer número indica el número de entradas y el segundo el número de salidas que es preciso que tenga el módulo para poder contar cuantos 1´s hay presentes en sus entradas. *Un CSA de n bits consiste en n C(3,2) que operan en paralelo y que no poseen interconexiones entre sus arrastres de salida.* Esta idea de los contadores se puede generalizar a un $C(n,m)$ donde $m = \lceil \log_2 n \rceil$ ($\lceil x \rceil$ representa el menor entero mayor o igual a x).

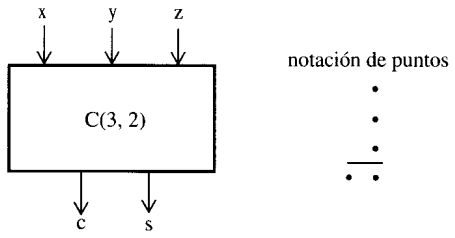


Figura 4.24: Diagrama de bloques y representación de puntos de un SBC

En la Figura 4.25 se muestra un CSA para la suma de 4 operandos x, y, z y w de 4 bits cada uno. Los dos niveles superiores son CSA´s de 4 bits, mientras que el tercer nivel es un sumador con propagación de los arrastres (SPA). Sin embargo este último nivel, para acelerar el proceso de la suma, se puede sustituir por un sumador con anticipación de los arrastres (SAA) o cualquier otro esquema de sumador de alta velocidad que se desee. Debe observarse que los bits de la suma parcial y los bits de los arrastres se interconectan de forma tal que se garantiza que sólo los bits con el mismo peso se suman por cualquiera de los contadores $C(3,2)$.

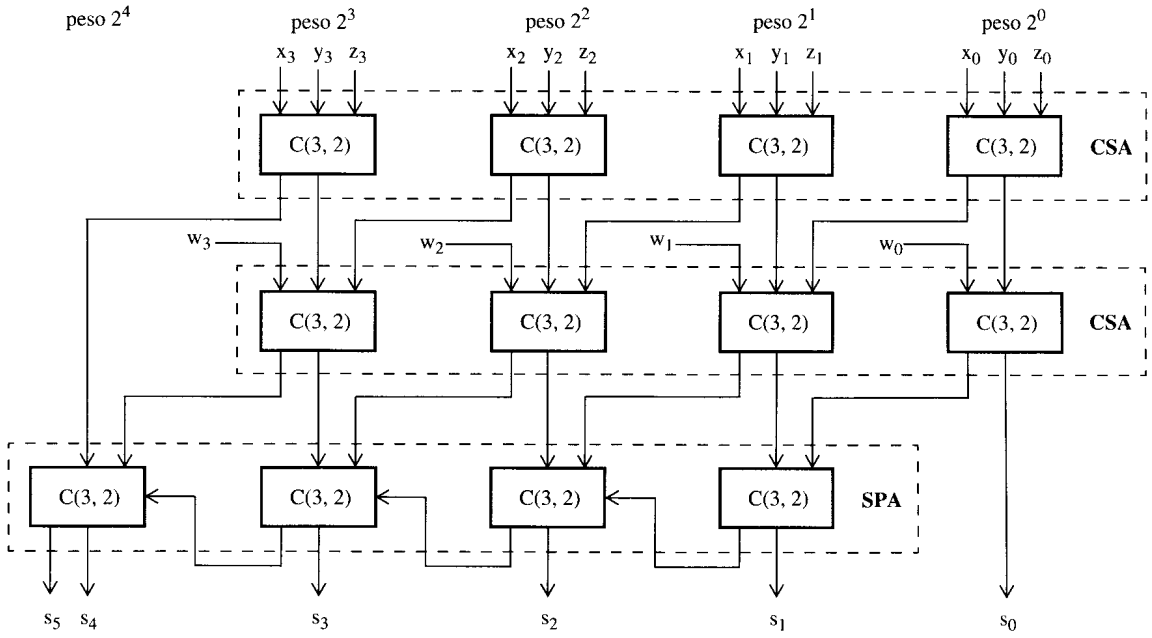


Figura 4.25: Un sumador CSA para 4 operandos de 4 bits

Para sumar los k operandos $x_1, x_2, \ldots\ldots, x_k$ se necesitan $(k - 2)$ unidades de CSA´s y un SAA. Si los CSA´s se disponen en cascada como en la Figura 4.25, entonces el tiempo para sumar los k operandos es:

$$\tau = (k-2)\tau_{CSA} + \tau_{SPA}$$

donde τ_{SPA} es el retardo de operación de un SPA y τ_{CSA} el retardo de un CSA que es igual según lo visto al de un SBC. El resultado final puede requerir una longitud de $n + \lceil \log_2 k \rceil$ bits, ya que la suma de k operandos de n bits cada uno puede ser tan grande como $(2^n - 1)k$.

Una forma mejor de organizar los CSA´s y reducir el tiempo de operación es emplear una estructura arborescente denominada *árbol de Wallace*. Este tipo de organización se estudiará en la sección 4-5 cuando se consideren los multiplicadores de alta velocidad.

La Figura 4.26 muestra como utilizar solamente un nivel de CSA para sumar un conjunto de números de n bits. En primer lugar se aplican tres números a las entradas x, y y z. La suma y el arrastre generado por estos tres números se realimentan a las entradas x e y que se suman con el cuarto número que se introduce en la entrada z. Este proceso continúa hasta que se suman todos los números.

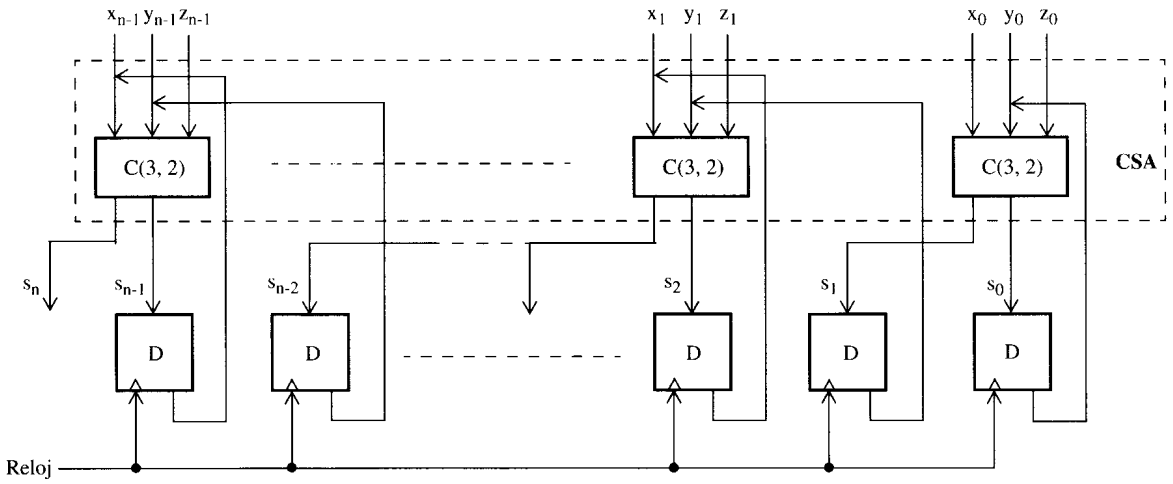


Figura 4.26: Diagrama de bloques de un CSA de n bits y de un sólo nivel que suma un conjunto de números

Sección 4-3

Sumadores en código BCD

Los sumadores que operan sobre números decimales codificados en binario, como el código BCD, son más complejos que los sumadores binarios. Los sumadores en código BCD se pueden diseñar como si fueran sumadores binarios, añadiéndoles unos circuitos de corrección que garanticen la correcta codificación de los resultados.

Si se suman dos dígitos BCD en un sumador binario de 4 bits el resultado es correcto si es menor que 10. El ejemplo siguiente muestra esta situación:

x = 2		x = 0 0 1 0	
y = 6		y = 0 1 1 0	
c = 0	s = 8	c = 0 s = 1 0 0 0	(correcto)

Sin embargo, cuando la suma es mayor o igual que 10, no es correcta porque no representa a ningún dígito codificado en BCD y debe ser corregida. Es fácil comprobar que la corrección consiste en añadir el dígito BCD 6 al resultado anterior. El ejemplo que se da a continuación corresponde a este caso.

x = 7		x = 0 1 1 1	
y = 8		y = 1 0 0 0	
c = 1	s = 5	c = 0 s = 1 1 1 1	(incorrecto)
		+ 0 1 1 0	(sumar 6)
		c = 1 s = 0 1 0 1	(correcto)

En la Figura 4.27 se muestra el circuito que realiza la suma de dos dígitos codificados en BCD. Los dos dígitos BCD's se aplican al primer sumador binario de 4 bits. El sumador realizará la suma en binario y producirá un resultado comprendido entre 0 y 19 (= 9 + 9 + 1 arrastre a la entrada del sumador). La salida de este sumador se representa por k, z_8, z_4, z_2 y z_1; k es el arrastre del sumador y los subíndices de z indican los pesos 8, 4, 2 y 1 que se asignan a los cuatro bits en el código BCD.

El circuito lógico que detecta cuándo es necesario corregir el resultado (c = 1) se obtiene de la forma siguiente:

a) Se requiere una corrección si hay un arrastre de salida en el primer sumador (k = 1). Esto sucede cuando la suma de los dos dígitos BCD es mayor que 15.

b) También se precisa corrección cuando la suma está comprendida entre 10 y 15 (c_1 = 1). Hay que detectar las 6 configuraciones que van desde $z_8\, z_4\, z_2\, z_1$ = 1010 a $z_8\, z_4\, z_2\, z_1$ = 1111.

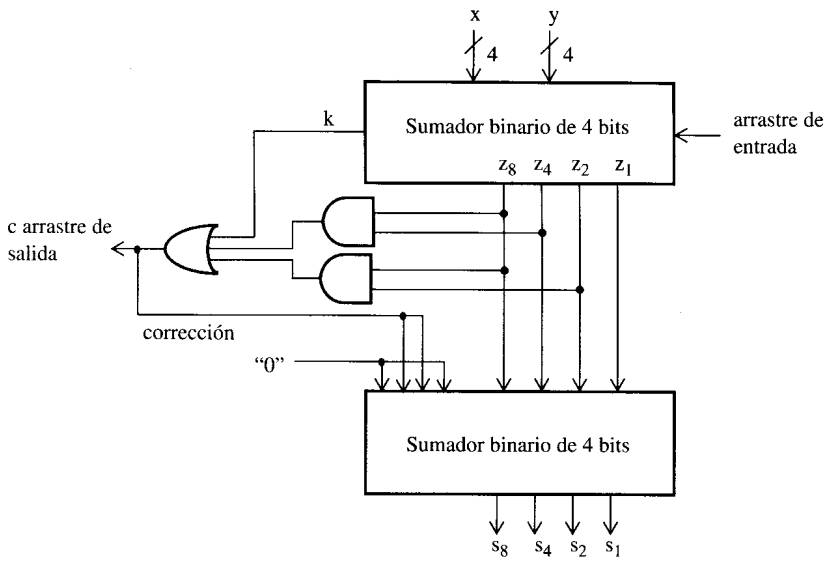


Figura 4.27: Sumador de dos dígitos en código BCD

El mapa de Karnaugh de la Figura 4.28 permite obtener la condición lógica (c_1) correspondiente.

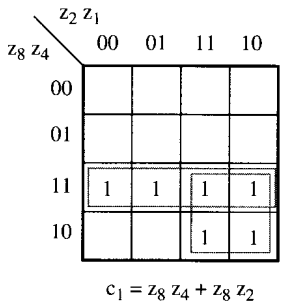


Figura 4.28: Corrección en la suma BCD

Teniendo en cuenta a) y b) se puede expresar la condición de corrección por la siguiente función booleana:

$$c = k + c_1 = k + z_8 z_4 + z_8 z_2$$

Cuando $c = 1$, es necesario añadir $6_{10} = 0110_2$ a la suma producida por el primer sumador y proporcionar un arrastre de salida para la siguiente etapa de dígitos BCD. El segundo sumador binario de 4 bits corrige el resultado añadiendo 6 cuando $c = 1$. Debe observarse que cuando $c = 0$, el segundo sumador no corrige el resultado dado por el primero, ya que lo que se le suma es cero.

4.3.1 Organización de los sumadores en código BCD

La suma de dos números de n dígitos decimales codificados en BCD, se puede hacer de tres formas:

a) *Sumador paralelo*

Contiene n sumadores BCD como el descrito en la Figura 4.27. El arrastre de salida de cada sumador, se conecta al arrastre de entrada del siguiente sumador (ver Figura 4.29). La suma se genera en paralelo, independientemente de la longitud de los operandos.

Para conseguir retardos pequeños en la propagación de los arrastres, de la misma forma que se vió al estudiar los sumadores binarios, los sumadores BCD's pueden incluir la lógica necesaria que permita la anticipación del arrastre.

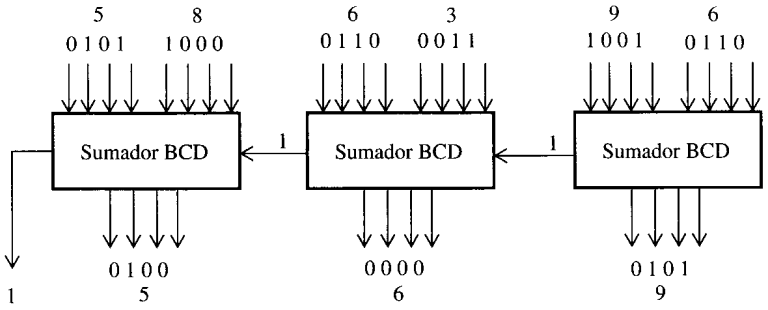


Figura 4.29: Sumador BCD paralelo

b) *Sumador dígito-serie, bit-paralelo*

En esta organización los dígitos se aplican de forma serie a un único sumador BCD, aunque los bits de cada dígito codificado se transfieren en paralelo (ver Figura 4.30).

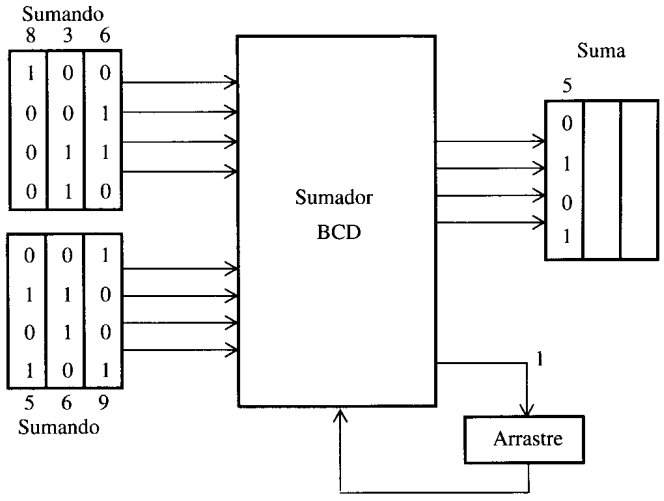


Figura 4.30: Sumador BCD dígito-serie, bit-paralelo

La suma se genera desplazando los números decimales, de uno en uno, a través del sumador BCD. Si son n los dígitos BCD's, se necesitarán n pasos para completar la operación de suma.

c) *Sumador dígito-serie, bit-serie*

Es la estructura de menor coste aunque también es la más lenta. En este caso, los bits se desplazan de uno en uno a través de un SBC (ver Figura 4.31). La suma binaria que se forma después de cuatro desplazamientos, hay que corregirla para obtener un dígito válido en código BCD.

Esta corrección, como ya se indicó, consiste en comprobar la suma binaria y si es mayor o igual a 10, sumar 6 y generar un arrastre para el siguiente par de dígitos.

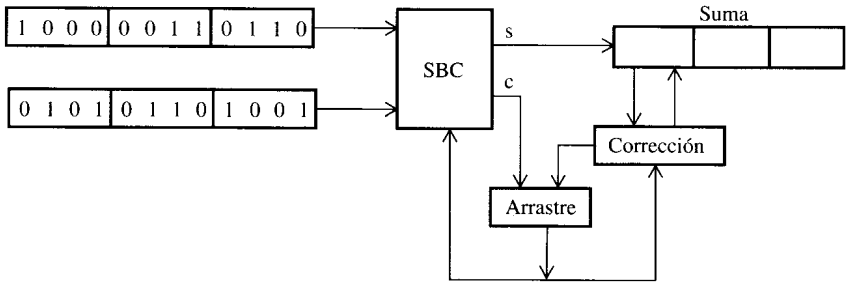


Figura 4.31: Sumador BCD dígito-serie, bit-serie

4.3.2 Restador en código BCD

La resta de dos números decimales codificados en BCD se puede realizar utilizando un circuito restador en código BCD, que aunque similar en su filosofía al sumador estudiado, es diferente. Una alternativa más económica consiste en formar el complemento a 9 ó 10 del sustraendo y sumarlo al minuendo (de manera similar al caso binario en el que se utilizaba el complemento a 1 ó a 2).

Como el código BCD no es un código autocomplementario, el complemento a 9 de un dígito no se obtiene complementando cada bit del código, se genera mediante un circuito que resta cada dígito BCD de 9. Esto se consigue complementando los bits en la representación codificada del dígito, a condición de que se incluya la corrección necesaria. Existen dos métodos para efectuar la corrección.

1) En el primer método, se suma el número 10 (1010) a cada dígito BCD complementado y no se considera el arrastre después de cada suma. En efecto, complementar cada bit de un número binario N de 4 bits es idéntico a restarlo del número 15 (1111). Si a continuación se le suma 10 se obtiene: $15 - N + 10 = 9 - N + 16$. Pero 16 representa el arrastre de salida que no se considera.

2) En el segundo método, se suma el número 6 (0110) a cada dígito BCD antes de complementarlo, es decir: $15 - (N + 6) = 9 - N$, que es el complemento a 9 de N.

También se puede obtener el complemento a 9 de un dígito BCD mediante un circuito combinacional que dispone de una señal de control M. La entrada a este circuito y_8, y_4, y_2 e y_1, es el dígito del sustraendo (o del segundo sumando) y su salida Y_8, Y_4, Y_2 e Y_1, es igual a la entrada cuando $M = 0$, e igual al complemento a 9 de la entrada cuando $M = 1$. Las ecuaciones lógicas del circuito combinacional "complementador a 9" son las siguientes (ver problema 15):

$$
\begin{aligned}
Y_1 &= y_1\,\overline{M} + \overline{y}_1 M \\
Y_2 &= y_2 \\
Y_4 &= y_4\overline{M} + (\overline{y}_4 y_2 + y_4\,\overline{y}_2)\,M \\
Y_8 &= y_8\overline{M} + \overline{y}_8\,\overline{y}_4\,\overline{y}_2 M
\end{aligned}
\tag{4.10}
$$

De estas ecuaciones se observa que $Y = y$ cuando $M = 0$. Si $M = 1$, las salidas de Y generan el complemento a 9 de y. Cuando se conecta este circuito a un sumador BCD, el resultado es una unidad aritmética decimal que suma o resta dos dígitos BCD (ver Figura 4.32). La operación de suma/resta está controlada por la señal M:

$$
\begin{aligned}
&\text{si} \quad M = 0 \quad \Rightarrow \quad s = x + y \\
&\text{si} \quad M = 1 \quad \Rightarrow \quad s = x + C9(y)
\end{aligned}
$$

donde C9(y) representa el complemento a 9 de y. Para números con n dígitos decimales se necesitarán n módulos sumadores/restadores. El arrastre de salida c_{i+1} de una etapa, debe conectarse al arrastre de entrada c_i de la etapa de orden superior.

La mejor forma de realizar la resta de dos números decimales es la siguiente:

a) Hacer $M = 1$

b) Hacer $c_{-1} = 1$ (c_{-1} es el arrastre de entrada de la primera etapa)

si se cumplen estas condiciones, la salida del circuito será:

$$s = x + C10(y) \quad \Leftrightarrow \quad s = x - y$$

si no se considera el arrastre de salida de la última etapa.

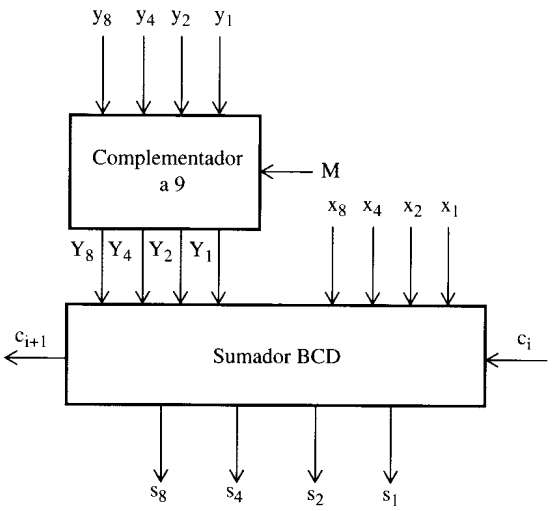


Figura 4.32: Sumador-restador de 2 dígitos en BCD

Sección 4-4

Multiplicadores binarios

La multiplicación es una operación compleja en comparación con la suma y la resta. Hay una gran variedad de algoritmos que permiten realizarla. El objetivo de esta sección es presentar los métodos más usuales, indicando sus ventajas e inconvenientes. Se comienza con la forma más simple de multiplicar dos números binarios sin signo (no negativos), es el procedimiento de multiplicación clásico de "lápiz y papel". Este método es muy ineficaz desde el punto de vista de velocidad de cálculo y coste del multiplicador, por lo que se propone una variante del mismo que evita alguno de sus problemas. Finalmente, para el caso de números con signo en complemento a 2 se presenta el algoritmo de Booth, que es el más clásico y el que ha dado lugar a un mayor número de generalizaciones.

4.4.1 Multiplicación de "lápiz y papel" de números sin signo

La Figura 4.33 muestra un ejemplo de multiplicación de dos números enteros binarios sin signo de 4 bits, tal como se efectúa utilizando el algoritmo clásico de la escuela de "lápiz y papel" (también se le conoce como algoritmo de "suma y desplazamiento"). *M* representa el multiplicando y *m* el multiplicador.

M =			1	0	0	1	Multiplicando (9)	
m =			1	0	1	1	Multiplicador (11)	
			1	0	0	1		
		1	0	0	1		Productos parciales	
	0	0	0	0				
1	0	0	1					
1	1	0	0	0	1	1	Producto (99)	

Figura 4.33: Multiplicación de dos números binarios sin signo

De este ejemplo se pueden hacer algunas consideraciones importantes:

1) La multiplicación genera 4 productos parciales p_i, uno por cada dígito del multiplicador. Estos productos parciales se suman después para obtener el producto final *P*.

2) Los productos parciales se definen fácilmente. Cuando el bit del multiplicador es 0, el producto parcial es 0. Cuando el multiplicador es 1, el producto parcial es el multiplicando:

$$p_i = M \times m_i$$

3) Cada producto parcial sucesivo se desplaza una posición a la izquierda en relación con el producto parcial precedente. El producto final *P* se obtiene sumando los productos parciales.

$$P = \sum_{i=0}^{n-1} p_i = M \sum_{i=0}^{n-1} m_i 2^i = M \times m$$

4) La multiplicación de dos números enteros binarios de n bits tiene una longitud de $2n$ bits.

En una versión estrictamente combinacional del algoritmo de lápiz y papel, los productos parciales se forman simultáneamente y se suman de forma concurrente. Cada bit del producto parcial $p_{ij} = m_i M_j$ se genera en una puerta AND que tiene como entradas un bit del multiplicador (m_i) y un bit del multiplicando (M_j) (ver Figura 4.34).

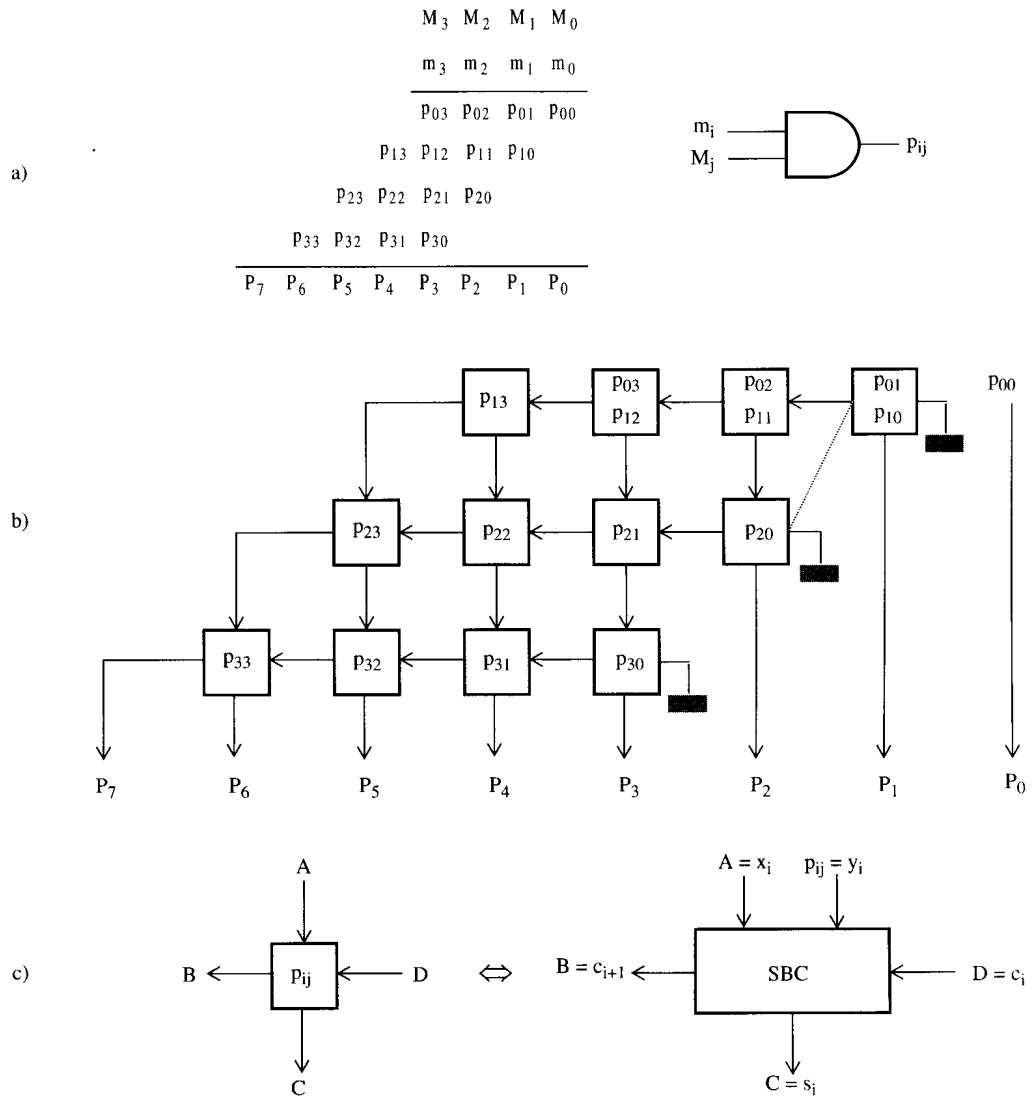


Figura 4.34: Multiplicador combinacional de 4 × 4 bits compuesto por 12 SBC's

La generación de los n productos parciales (p_i) y su suma para formar el producto final (P), se realiza en una red de $n(n-1)$ SBC's (12 para $n = 4$), tal como se muestra en la Figura 4.34. La estructura del multiplicador corresponde al algoritmo de suma y desplazamiento que se acaba de ver, porque cuando un bit del multiplicador vale 0 resulta un producto parcial de cero, y cuando vale 1 hace que el producto parcial sea el valor del multiplicando. El desplazamiento viene dado por la forma de interconectar los SBC's.

La velocidad de este multiplicador combinacional está determinada por el retardo en la propagación de los arrastres, es decir, el retardo desde p_{01} hasta p_{13} y desde p_{13} hasta P_7. Este camino está constituido por $2n-2 = 6$ etapas, cada una con el retardo propio de un SBC (se supone un retardo de 4 para cada SBC), lo que da un retardo de 24 al que hay que sumarle un retardo adicional correspondiente a la matriz AND de los bits p_{ij}. Se obtiene así un retardo total en la propagación de los arrastres de 25.

4.4.2 Mejoras en el algoritmo de "lápiz y papel"

Si se analiza críticamente el algoritmo anterior se observa que hay algunas cosas que se pueden modificar para conseguir una multiplicación más eficaz. En concreto se consideran las siguientes:

1) Se puede realizar la suma de los productos parciales tan pronto como se producen, en lugar de tener que esperar hasta el final. Esto elimina la necesidad de almacenar todos los productos parciales y en consecuencia se necesitan menos registros intermedios.

2) Se puede ahorrar tiempo en la generación de los productos parciales. Para cada 1 en el multiplicador se requieren las operaciones de suma y desplazamiento; pero para cada 0 sólo es necesario el desplazamiento.

La Figura 4.35 muestra el esquema de un multiplicador que tiene en cuenta estas modificaciones. El multiplicador y el multiplicando se cargan en dos registros de n bits (m y M). Se necesita un tercer registro A de n bits (registro acumulador) que inicialmente se carga a 0. Hay un registro C de 1 bit (registro de rebose) que se inicializa a 0 y que almacenará un posible bit de arrastre c_a que resulta de la suma. La operación del multiplicador se realiza de la forma siguiente:

a) La unidad de control, lee los bits del mutiplicador uno a uno.

b) Si m_0 es 1, el multiplicando M se suma con el contenido del registro A y el resultado de la suma se almacena en A.

c) Todos los bits de los registros C, A y m se desplazan simultáneamente 1 bit a la derecha, de la forma siguiente: $0 \rightarrow C$, $C \rightarrow A_{n-1}$, $A_{n-1} \rightarrow A_{n-2}$, ..., $A_0 \rightarrow m_{n-1}$, ..., $m_1 \rightarrow m_0$ y m_0 se pierde.

d) Si m_0 es 0, no se realiza la suma, sólo el desplazamiento.

e) Estas operaciones se repiten para cada bit del multiplicador m original. El producto resultante P, de $2n$ bits, estará contenido al final de este proceso en los registros A y m.

Esta forma de realizar el producto se puede expresar matemáticamente como sigue:

$$p_0 = 0$$

$$p_{i+1} = 2^{-1} (p_i + 2^n M \times m_i) \; i = 0,....., n-1 \qquad (4.11)$$

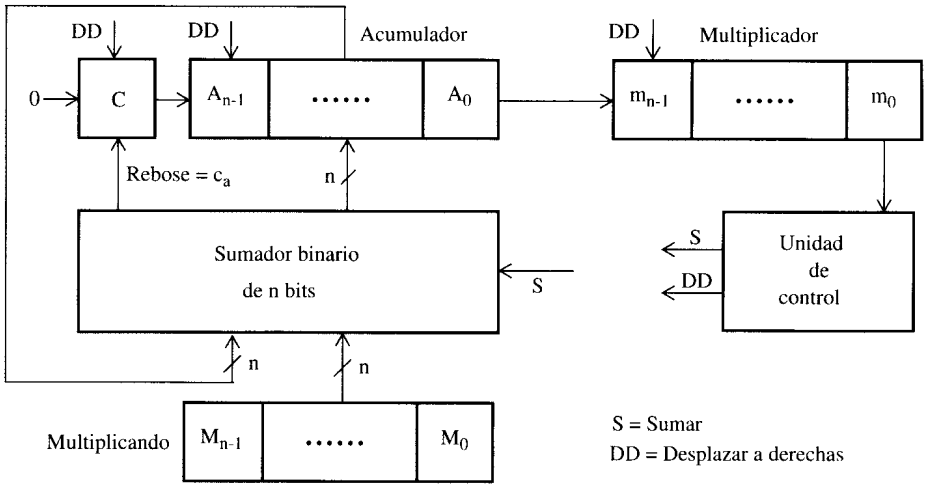


Figura 4.35: Esquema de un multiplicador que mejora el algoritmo de "lapiz y papel"

Se observa que cada paso consiste en la multiplicación de M por un bit m_i del multiplicador para formar Mm_i, seguido por una suma de dos operandos y por el desplazamiento de un bit a la derecha (2^{-1}). El factor 2^n que multiplica a $M \times m_i$ indica que el multiplicando M se tiene que alinear con la mitad más significativa del producto parcial que se encuentra almacenada en el registro A.

Este tipo de recurrencia, que utiliza desplazamientos a derecha en lugar de los desplazamientos a izquierda del algoritmo de "lápiz y papel", permite que la multiplicación pueda progresar desde el bit menos significativo del multiplicador mientras que el multiplicando retiene la misma posición en relación con el sumador. Esto se ilustra en la Figura 4.36 con el mismo ejemplo numérico que el empleado en el algoritmo de "lápiz y papel".

C	A	m			
0	0000	1011	Valores iniciales		
0	1001	1011	S	1er ciclo	
0	0100	1101	DD		
0	1101	1101	S	2° ciclo	
0	0110	1110	DD		
0	0011	0111	DD	3er ciclo	
0	1100	0111	S	4° ciclo	(Producto en A, m)
0	0110	0011	DD		

Figura 4.36: Ejemplo numérico del algoritmo de "lapiz y papel mejorado" (M contiene 1001)

En la Figura 4.37 se da un diagrama de flujo de la multiplicación de números binarios sin signo utilizando este algoritmo. Se ha utilizado la notación || para definir un registro compuesto constituido a partir de otros registros. Así por ejemplo $C \| A \| m$ representa un único registro obtenido al combinar en ese orden

los registros C, A y m (el bit más significativo está en C y el menos significativo a la derecha de m). El símbolo >> especifica un desplazamiento del dato hacia la derecha, indicándose a la derecha del símbolo el número de bits de desplazamiento. Al sumar el contenido de los registros A y M se puede producir un arrastre en la operación que se representa por c_a. P es un contador que controla el número de iteraciones del bucle.

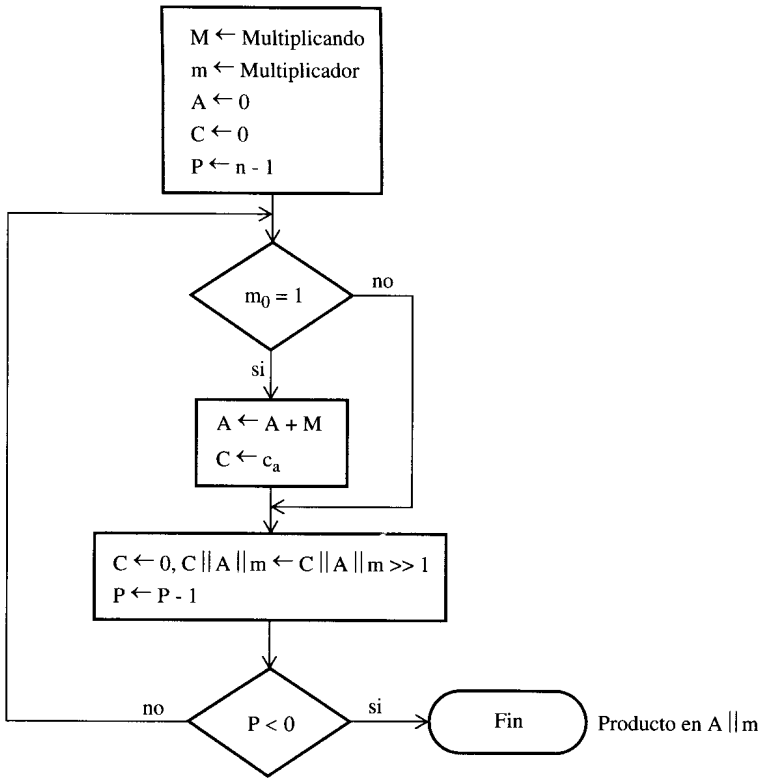


Figura 4.37: Diagrama de flujo de la multiplicación de números binarios sin signo

4.4.3 Multiplicación en complemento a 2: Algoritmo de Booth

Se ha visto ya que la suma y la resta se pueden realizar, en el caso de numeros con signo representados en complemento a 2, de la misma forma que si fuesen números sin signo. Considérese la siguiente suma:

$$\begin{array}{r} 1\,0\,1\,1 \\ +\,0\,0\,1\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

si estos dos números se consideran sin signo, se está sumando $11_{10} = 1011_2$ con $2_{10} = 0010_2$ para obtener $13_{10} = 1101_2$. Si los números están representados en complemento a 2, lo que se suma es $-5_{10} = 1011_2$ y $2_{10} = 0010_2$ y da como resultado $-3_{10} = 1101_2$. Desafortunadamente, este esquema tan sencillo no va bien con la multiplicación. Para verlo considérese otra vez la Figura 4.33 en la que se multiplica $9_{10} = 1001_2$ por

$11_{10} = 1011_2$ y da como resultado $99_{10} = 1100011_2$. Si se interpretan estos números en complemento a 2, se tiene $-7_{10} \times -5_{10} = -29_{10}$ ($1001_2 \times 1011_2 = 1100011_2$). Este ejemplo demuestra que la multiplicación realizada directamente produce resultados erróneos si el multiplicando y el multiplicador son números negativos. De hecho, tampoco se obtiene el resultado correcto si sólo uno de los dos es negativo. Para explicar esto conviene recordar como se puede expresar un número binario como una suma de potencias de 2. Así

$$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 2^3 + 2^1 + 2^0$$

Más aún, la multiplicación de un número binario por 2^n se efectúa desplazando ese número n bits a la izquierda. Teniendo esto en cuenta, en la Figura 4.38 se reconstruye la Figura 4.33 para hacer explícita la generación de los productos parciales en la multiplicación. La única diferencia en la Figura 4.38 es que se pone de manifiesto, que los productos parciales deben considerarse como números de $2n$ bits que se generan a partir de un multiplicando de n bits.

M = 1001	
m = 1011	
00001001	$1001 \times 1 \times 2^0$
00010010	$1001 \times 1 \times 2^1$
00000000	$1001 \times 0 \times 2^2$
01001000	$1001 \times 1 \times 2^3$
01100011	

Figura 4.38: La multiplicación de 2 enteros de 4 bits sin signo da un resultado de 8 bits

Así, si el multiplicando de 4 bits 1001 (considerado como un entero sin signo) se almacena en una palabra de 8 bits como 00001001, cada producto parcial (salvo el que corresponde a 2^0) consiste en este número desplazado a la izquierda con las posiciones no ocupadas a su derecha rellenas con ceros (por ejemplo, un desplazamiento de dos lugares a la izquierda da 00100100).

Se puede ver ahora que la multiplicación realizada de forma directa no funciona si el multiplicando es negativo. El problema es, que cada contribución del multiplicando negativo como un producto parcial debe ser un número negativo en un campo de $2n$ bits; los bits de signo de los productos parciales tienen que estar alineados. Esto puede observarse en la Figura 4.39a que muestra la multiplicación de 1001 por 0101. Si estos números se tratan como enteros sin signo, la multiplicación de $9 \times 5 = 45$ se obtiene correctamente. Sin embargo, si 1001 se interpreta como el número -7 en complemento a 2, entonces cada producto parcial debe ser un número negativo en complemento a 2 tal como se muestra en la Figura 4.39b. En este caso el resultado correcto se consigue rellenando cada producto parcial a su izquierda con unos.

1001	(9)
× 0101	(5)
00001001	$(1001) \times 2^0$
00100100	$(1001) \times 2^2$
00101101	(45)

(a) Enteros sin signo

1001	(−7)
× 0101	(5)
11111001	$(-7) \times 2^0 = (-7)$
11100100	$(-7) \times 2^2 = (-28)$
11011101	(−35)

(b) Enteros en complemento a 2

Figura 4.39: Comparación de la multiplicación de enteros sin signo y en complemento a 2

Si el multiplicador es negativo el resultado de la multiplicación tampoco es correcto. La razón es que los bits del multiplicador no se corresponden ya con los desplazamientos o multiplicaciones que hay que realizar. Por ejemplo:

$$-5 = 1011 = -(0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -2^2 - 2^0$$

por lo que el multiplicador no se puede utilizar directamente de la forma que se ha descrito. Para superar este problema se tienen varias alternativas.

La más directa sería convertir a números positivos tanto el multiplicando como el multiplicador, realizar la multiplicación y formar el complemento a 2 del resultado si y sólo si difieren los signos de los dos números originales. A pesar de la sencillez conceptual de este método, es mejor utilizar otras técnicas que no necesitan el paso final de complementar a 2 el resultado. Uno de los métodos más utilizado es el *algoritmo de Booth*, que también tiene la ventaja de acelerar la multiplicación como se verá a continuación. En la Figura 4.40 se da un diagrama de flujo del algoritmo de Booth. Merecen destacarse las siguientes consideraciones:

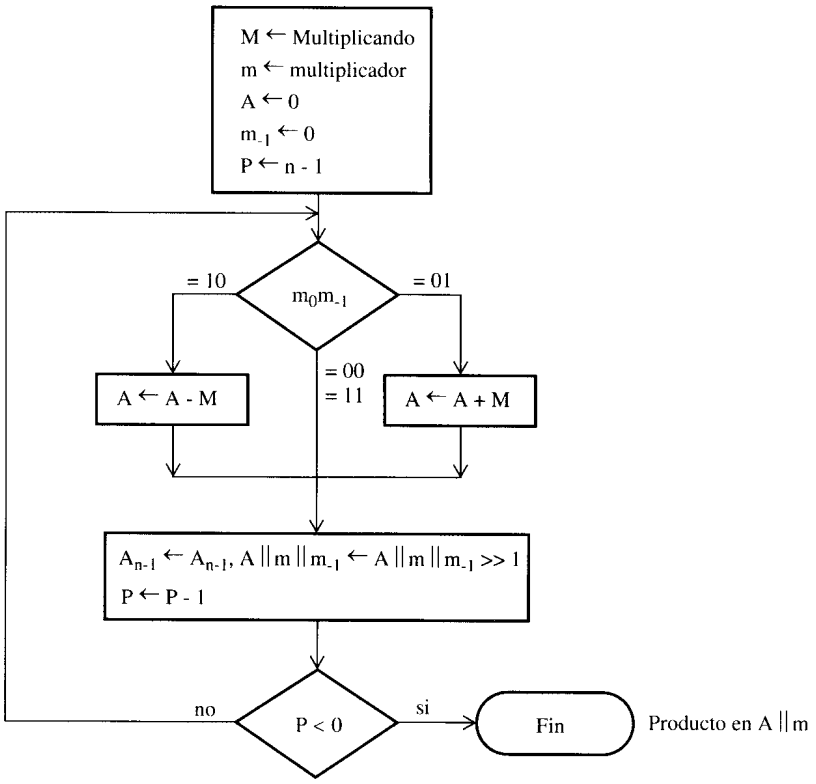


Figura 4.40: Diagrama de flujo del algoritmo de Booth

a) El multiplicando y el multiplicador se almacenan en los registros *M* y *m* respectivamente.

b) Hay un registro de 1 bit que se coloca a la derecha del bit menos significativo (m_0) del registro *m*. Se representan por m_{-1} y su utilidad se explica posteriormente.

c) El resultado de la multiplicación se obtiene en los registros A y m. Los registros A, m y m_{-1} se inicializan a 0.

d) La unidad de control del multiplicador examina uno a uno los bits del registro m. Sin embargo, en el algoritmo de Booth cuando se examina cada bit, se comprueba también el bit que está a su derecha.

e) Si los dos bits son iguales (1-1 ó 0-0), todos los bits de los registros A, m y m_{-1} se desplazan un bit a la derecha.

f) Si los dos bits difieren, el multiplicando se suma o se resta del registro A, según que los dos bits sean 0-1 ó 1-0 respectivamente.

g) El desplazamiento se produce siempre después de la suma o la resta. En cualquiera de los casos, el desplazamiento a derecha es tal que el bit más significativo de A, es decir A_{n-1}, permanece en A_{n-1} y se desplaza a A_{n-2}. Esto se necesita para preservar el signo del número en A y en m. Este desplazamiento se conoce como *desplazamiento aritmético*, ya que conserva el bit de signo (las operaciones de desplazamiento se verán en la sección 4-9).

En la Figura 4.41 se muestra la secuencia de pasos del algoritmo de Booth para multiplicar 6 por 3.

A	m	m_{-1}			
0 0 0 0	0 0 1 1	0	Valores iniciales		
1 0 1 0	0 0 1 1	0	A ← A − M		
1 1 0 1	0 0 0 1	1	Desplazar	1er ciclo	
1 1 1 0	1 0 0 0	1	Desplazar	2° ciclo	
0 1 0 0	1 0 0 0	1	A ← A + M		
0 0 1 0	0 1 0 0	0	Desplazar	3er ciclo	
0 0 0 1	0 0 1 0	0	Desplazar	4° ciclo	(Producto en A || m)

Figura 4.41: Ejemplo del algoritmo de Booth (M contiene 0110)

En la Figura 4.42a se representa la misma operación de forma más compacta. El resto de la Figura 4.42 muestra otros ejemplos del algoritmo que, como se puede ver, funciona correctamente con cualquier combinación de números positivos y negativos. Es de destacar la eficacia del algoritmo de Booth, ya que ahorra las sumas cuando hay bloques de 1's ó 0's, con un promedio de una suma o resta por bloque.

A continuación se justifica por qué funciona correctamente el algoritmo de Booth. En primer lugar se considera el caso en que el multiplicador m es positivo. En particular, se supone que m consiste en una secuencia de 1's rodeada por 0's, por ejemplo 00111100. Como se sabe, la multiplicación se puede realizar sumando copias del multiplicando M desplazadas adecuadamente:

$$M \times (00111100) = M \times (2^5 + 2^4 + 2^3 + 2^2)$$
$$M \times (32 + 16 + 8 + 4) = M \times 60$$

(a) (6) × (3) = (18)

	0 1 1 0 × 0 0 1 1 (0)	
	1 1 1 1 1 0 1 0	1 - 0
	0 0 0 0 0 0 0	1 - 1
	0 0 0 1 1 0	0 - 1
	0 0 0 1 0 0 1 0	(18)

(b) (6) × (−3) = (−18)

	0 1 1 0 × 1 1 0 1 (0)	
	1 1 1 1 1 0 1 0	1 - 0
	0 0 0 0 1 1 0	0 - 1
	1 1 1 0 1 0	1 - 0
	1 1 1 0 1 1 1 0	(-18)

(c) (-6) × (3) = (−18)

	1 0 1 0 × 0 0 1 1 (0)	
	0 0 0 0 0 1 1 0	1 - 0
	0 0 0 0 0 0 0	1 - 1
	1 1 1 0 1 0	0 - 1
	1 1 1 0 1 1 1 0	(-18)

(d) (-6) × (−3) = (18)

	1 0 1 0 × 1 1 0 1 (0)	
	0 0 0 0 0 1 1 0	1 - 0
	1 1 1 1 0 1 0	0 - 1
	0 0 0 1 1 0	1 - 0
	0 0 0 1 0 0 1 0	(18)

Figura 4.42: Diferentes ejemplos del algoritmo de Booth

el número de estas operaciones se puede reducir a dos si se observa que:

$$2^j + 2^{j-1} + ...+ 2^{j-k} = 2^{j+1} - 2^{j-k} \tag{4.12}$$

teniendo esto en cuenta, el resultado anterior se puede expresar como:

$$M \times (00111100) = M \times (2^6 - 2^2) = M \times (64 - 4) = M \times 60$$

En el ejemplo se ve que el producto se puede generar mediante una suma y una resta del multiplicando. El esquema se generaliza sin ninguna dificultad a cualquier número de secuencias de 1's en el multiplicador, incluyendo el caso de un único 1 que se trata también como una secuencia. Así

$$M \times (01110010) = M \times (2^7 + 2^6 + 2^5 + 2^1) = M \times (2^8 - 2^5 + 2^2 - 2^1)$$

El algoritmo de Booth realiza precisamente este esquema, efectuando una resta cuando se encuentra el primer 1 de una secuencia (1-0) y una suma al detectar el final de la secuencia (0-1). Para demostrar que este mismo principio funciona cuando el multiplicador es negativo, se necesita observar lo siguiente: sea *X* un número negativo representado en complemento a 2, es decir:

$$\text{Representación de } X = \{1\ x_{n-2}\ x_{n-3} ....x_1\ x_0\}$$

el valor de *X* se puede expresar como sigue:

$$X = -2^{n-1} + x_{n-2} \times 2^{n-2} + ...+ x_1 \times 2^1 + x_0 \times 2^0 \tag{4.13}$$

Como *X* es negativo, su bit más significativo es 1. Sin pérdida de generalidad se supone que el 0 más a la izquierda en *X* está en la posición *k*-ésima. Así *X* es de la forma:

$$\text{Representación de } X = \{1\ 1 .... 1\ 0\ x_{k-1}\ x_{k-2} ....x_1\ x_0\} \tag{4.14}$$

su valor será:

$$X = -2^{n-1} + 2^{n-2} + ...+ 2^{k+1} + x_{k-1} \times 2^{k-1} + ...+ x_0 \times 2^0 \tag{4.15}$$

teniendo en cuenta la ecuación (4.12), se puede ver que

$$2^{n-2} + 2^{n-3} + ... + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

reordenando los términos de esta última expresión:

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^{k+1} = -2^{k+1} \tag{4.16}$$

sustituyendo la ecuación (4.16) en la ecuación (4.15), se obtiene:

$$X = -2^{k+1} + x_{k-1} \times 2^{k-1} + ... + x_0 \times 2^0 \tag{4.17}$$

Si se tiene en cuenta el algoritmo de Booth y la representación de X (ecuación (4.14)), es evidente que todos los bits de X hasta el 0 que está más a la izquierda se manejan adecuadamente, puesto que producen todos los términos en la ecuación (4.17) salvo (-2^{k+1}) y están así en la forma apropiada. Corresponde pues al primer caso visto de multiplicador positivo, si se considera el número hasta el 0 más a la izquierda como un número positivo. Cuando el algoritmo de Booth examina los bits que están más a la izquierda del 0 de la posición k-ésima, y encuentra el siguiente 1 (2^{k+1}), ocurre una transición 1-0 y tiene lugar una resta (-2^{k+1}). Éste es precisamente el término que quedaba en la ecuación (4.17).

4.4.4 Ejemplo: Algoritmo de Booth

Se considera la multiplicación de $M \times (-6)$. En la representación complemento a 2 y utilizando una longitud de palabra de 8 bits, $-6_{10} = 11111010_2$. De la ecuación (4.13) se sabe que:

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

por lo que

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

utilizando la ecuación (4.17) hasta el primer 0 más a la izquierda, la expresión anterior se puede expresar como:

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

Finalmente, siguiendo la misma línea de razonamiento con el segundo 0 se obtiene:

$$M \times (11111010) = M \times (-2^3 + 2^2 - 2^1)$$

Se puede ver que el algoritmo de Booth se adecúa a este esquema. Realiza una resta cuando se encuentra el primer 1 (transición 1-0), una suma cuando detecta una transición 0-1 y finalmente otra resta cuando ve el primer 1 de la siguiente secuencia de 1's. De esta forma el algoritmo de Booth efectúa menos sumas y restas que si se hubiese utilizado un algoritmo directo.

Sección 4-5

Multiplicadores de alta velocidad

En esta sección se analizan dos métodos básicos para acelerar la multiplicación. El primero genera más rápidamente la suma de los productos parciales utilizando sumadores del tipo de arrastre almacenado y una estructura de interconexión denominada "árbol de Wallace". El segundo reduce el número de productos parciales empleando un algoritmo de Booth modificado.

4.5.1 Suma rápida de los productos parciales

El elevado retardo en el multiplicador combinacional de la Figura 4.34 se debe a la propagación del arrastre a lo largo de todas las etapas. Este retardo se puede reducir casi a la mitad haciendo un cambio sencillo en la estructura de interconexión de los arrastres de los SBC's (que se indica en la Figura 4.34 por la línea discontinua).

En lugar de esperar a que el arrastre se propague a lo largo de todos los SBC's que se encuentran en su misma fila a su izquierda, se suma a la entrada de arrastre del SBC que está localizado en la fila de debajo y a su izquierda. De esta forma se aplaza la suma de los arrastres. El principio de posponer la suma de los arrastres se puede extender a todas las etapas del sumador excepto a la última. Los arrastres que se obtienen forman un operando de n bits, que se añade a la suma parcial de la última etapa en un sumador rápido dotado con aceleración de arrastre.

Esta técnica también permite utilizar una tercera entrada en cada uno de los SBC's de la etapa superior (primera fila), de manera que los tres primeros productos parciales se pueden sumar en la primera etapa. Dicha modificación reduce el número de etapas del sumador de n-1 a n-2. El esquema de esta estructura para $n = 4$ se muestra en la Figura 4.43. Se observa que corresponde al tipo de sumadores con arrastre almacenado o sumadores CSA que se estudiaron anteriormente en el apartado 4.2.7.

Utilizando un árbol de Wallace se puede aumentar aún más la velocidad del multiplicador. El árbol de Wallace es una interconexión de SBC's con arrastre almacenado que reduce n productos parciales a dos operandos. A estos sumadores se les denomina por su acrónimo en inglés: CSA (Carry Save Adder- sumador de arrastre almacenado).

El principio del árbol de Wallace es utilizar un CSA para reducir tres bits de igual peso a dos bits, uno de suma y otro de arrastre. En una multiplicación de $n \times n$ se generan n productos parciales que se pueden considerar como columnas adyacentes de bits con igual peso.

La altura máxima de una columna es n bits que se divide en grupos de tres bits. Estos grupos se pueden reducir simultáneamente a dos bits, lo que resulta en una nueva columna con una altura de $(2/3)n$ bits. La nueva columna generada se particiona otra vez en grupos de tres bits, y el proceso se repite hasta que la altura de la columna final es de dos bits.

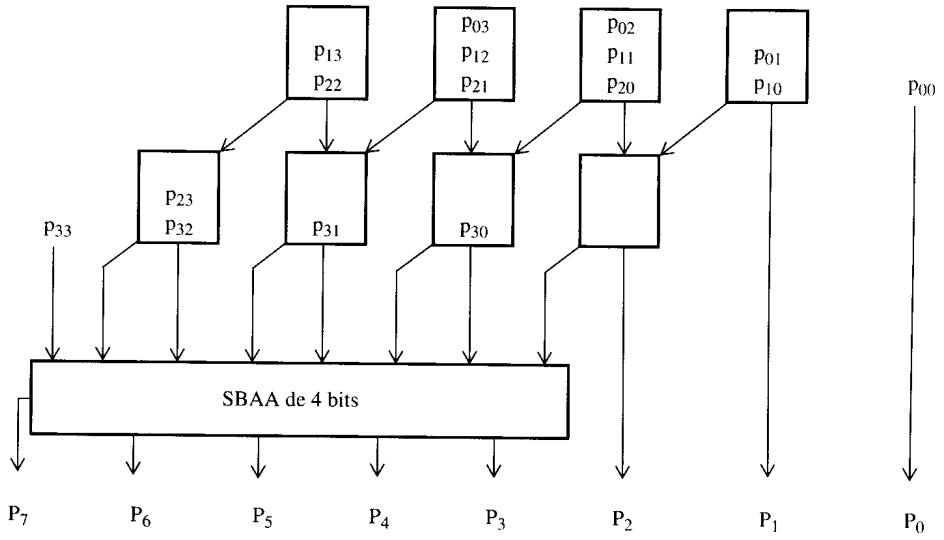


Figura 4.43: Multiplicador de 4 × 4 bits compuesto por CSA's y un SBAA de 4 bits

Para visualizar la interconexión del árbol de Wallace, en la Figura 4.44 se da una representación de puntos de un multiplicador de 8 × 8.

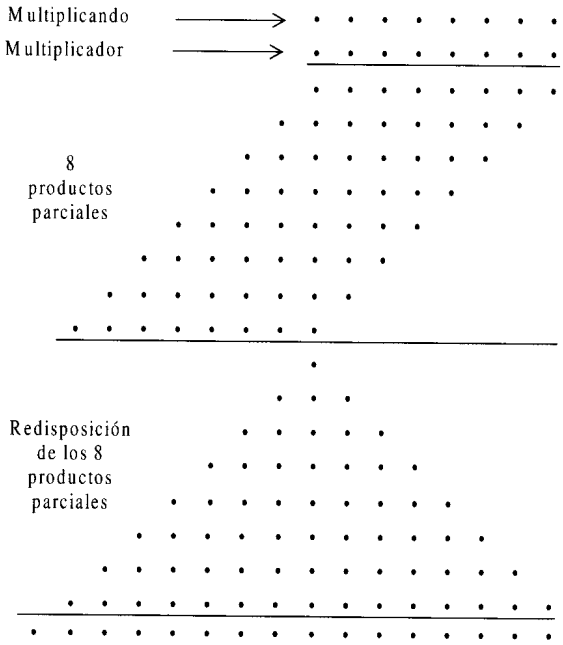


Figura 4.44: Representación de puntos de un multiplicador de 8 × 8 bits

En la Figura 4.45 se muestra una reducción del árbol de Wallace. Son necesarios 4 niveles de CSA's para reducir los 8 productos parciales a 2 operandos (2 niveles menos que si se hubiese utilizado el esquema de la Figura 4.43). El retardo de propagación del multiplicador de la Figura 4.45 es de 23 y se compone de los siguientes elementos:

1) Un retardo de 1 para generar los productos parciales.

2) Un retardo de 16 producido por 4 niveles de CSA's.

3) Un retardo de 6 producido en un sumador con anticipación de arrastres de 2 niveles.

En general para un árbol de Wallace el número de niveles de CSA's necesarios para reducir los n productos parciales a 2 viene dado por:

$$k = \left\lceil \frac{\log_2 n - 1}{\log_2 3 - 1} \right\rceil \tag{4.18}$$

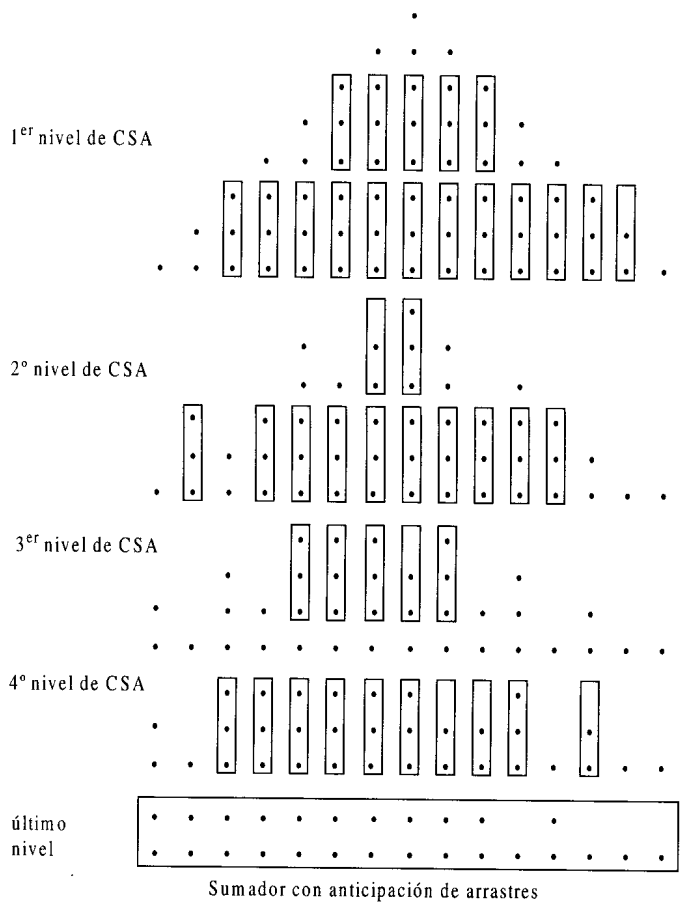


Figura 4.45: Reducción de la matriz de productos parciales de un multiplicador 8 × 8 utilizando sumadores del tipo CSA

donde $\lceil x \rceil$ es el menor entero mayor o igual a x. Así, en una multiplicacion de 32 × 32 bits, se necesitan 8 niveles de CSA's para reducir los 32 productos parciales a 2 operandos, en lugar de los 30 niveles de CSA's empleados cuando se utiliza el esquema de la Figura 4.45.

4.5.2 Algoritmo de Booth modificado

Otra técnica para acelerar la multiplicación consiste en una modificación del algoritmo de Booth, que permite reducir a la mitad el número de productos parciales (es decir, el número de niveles de CSA's) y duplicar por tanto la velocidad del multiplicador. El objetivo del algoritmo de Booth era examinar secuencias de bits en lugar de formar un producto parcial para cada bit. Cuando la secuencia es de 0's no hay ningún problema. Si la secuencia es de 1's, el algoritmo de Booth utiliza una propiedad especial de estas secuencias que viene dada por la expresión (4.12), de manera que el valor de dicha secuencia se obtiene restando el peso del 1 que está más a la derecha del módulo de la cadena; por ejemplo, la secuencia 1111 se calcula como $2^4 - 2^0 = 15$ y la secuencia 11100 se calcula como $2^5 - 2^2 = 28$. En el algoritmo de Booth modificado, se divide el multiplicador en subcadenas de 3 bits cada una y las 8 posibles combinaciones se calculan según la Tabla 4.12 .

2^1 m_{i+1}	2^0 m_i	2^{-1} m_{i-1}	Comentario
0	0	0	Sumar cero (no hay secuencia)
0	0	1	Sumar M (fin de secuencia)
0	1	0	Sumar M (secuencia)
0	1	1	Sumar 2M (fin de secuencia)
1	0	0	Restar 2M (comienzo de secuencia)
1	0	1	Restar M (-2M + M)
1	1	0	Restar M (comienzo de secuencia)
1	1	1	Restar cero (centro de la secuencia)

Tabla 4.12: Decodificación de 3 bits simultáneos del multiplicador

El algoritmo de Booth modificado requiere que el multiplicador se amplíe con un 0 a la derecha del bit menos significativo, y que se extienda el bit de signo a su izquierda, siempre que sea necesario, para formar el último grupo de 3 bits. Así, si el multiplicador tiene 8 bits, se divide en 5 grupos de 3 bits cada uno. Los grupos adyacentes tienen un bit en común (ver Figura 4.46).

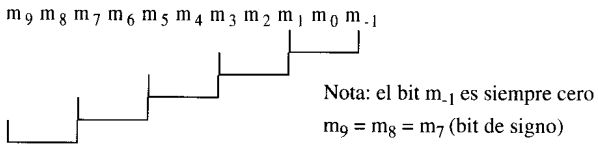


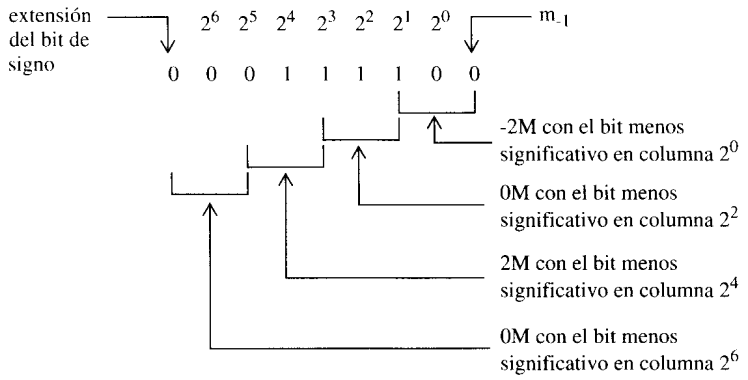
Figura 4.46: Partición del multiplicador ampliado en 5 grupos de 3 bits

El peso del bit del centro de cada grupo corresponde a la columna donde hay que empezar a colocar el producto parcial correspondiente. Se ve pues, que el algoritmo de Booth modificado incorpora un esquema de codificación del multiplicador que permite cada vez desplazamientos constantes de 2 bits, aunque se examinan 3 bits del multiplicador. En un multiplicador de 8 × 8 esto da 5 productos parciales, en lugar de los 8 que se necesitan sin codificación. Para justificar las entradas de la Tabla 4-10 hay que tener presente como opera el algoritmo de Booth :

- Se examinan los bits del multiplicador de derecha a izquierda, y se efectúa una resta del multiplicando cuando se encuentra el primer 1 de una secuencia (1-0) y una suma al detectar el final de la secuencia (0-1). Por ejemplo sea:

$$M \times (0011110) = M \times (2^4 + 2^3 + 2^2 + 2^1) = M \times (2^5 - 2^1)$$

En la Figura 4.47 se muestra, para este ejemplo, cómo se aplican las entradas de la Tabla 4.12 y se justifica que se obtiene el mismo resultado que aplicando el algoritmo de Booth.



El resultado de esta secuencia es: $-2M \times 2^0 + 2M \times 2^4 = M \times (2^5 - 2^1)$ que coincide con el obtenido aplicando el algoritmo de Booth

Figura 4.47: Ejemplo de aplicación del algoritmo de Booth modificado

A pesar de estas ventajas, el algoritmo de Booth modificado presenta un problema: se necesita una resta que se realiza, tal como se explicó en el apartado 4.1.5, sumando el C2 del número que se va a restar. Para formar el complemento a 2 de un número, en primer lugar se calcula el C1 (complementando cada bit del número) y a continuación se suma 1 al bit menos significativo. En la Tabla 4.12 se observa que cuando el valor del bit m_{i+1} es igual a 1 es necesario efectuar una resta, por lo que se puede utilizar este bit como señal de control M en el circuito sumador-restador de la Figura 4.12.

Un método rápido de obtener, de forma serie, el complemento a 2 de un número es el siguiente: se van examinando los bits del número a complementar de derecha a izquierda y se dejan inalterados hasta la aparición del primer 1, que queda inalterado, a partir de ese instante se complementan todos los bits. La Figura 4.48 muestra un ejemplo de aplicación de esta regla.

En complemento a 2 el bit de signo se debe extender hasta la longitud completa del resultado final. En la Figura 4.49 se muestran diferentes ejemplos numéricos del algoritmo de Booth modificado, para algunas combinaciones de números positivos y negativos en el multiplicando y en el multiplicador.

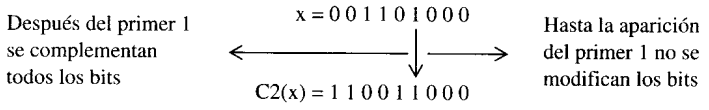


Figura 4.48: Forma rápida de obtener el complemento a 2 de un número

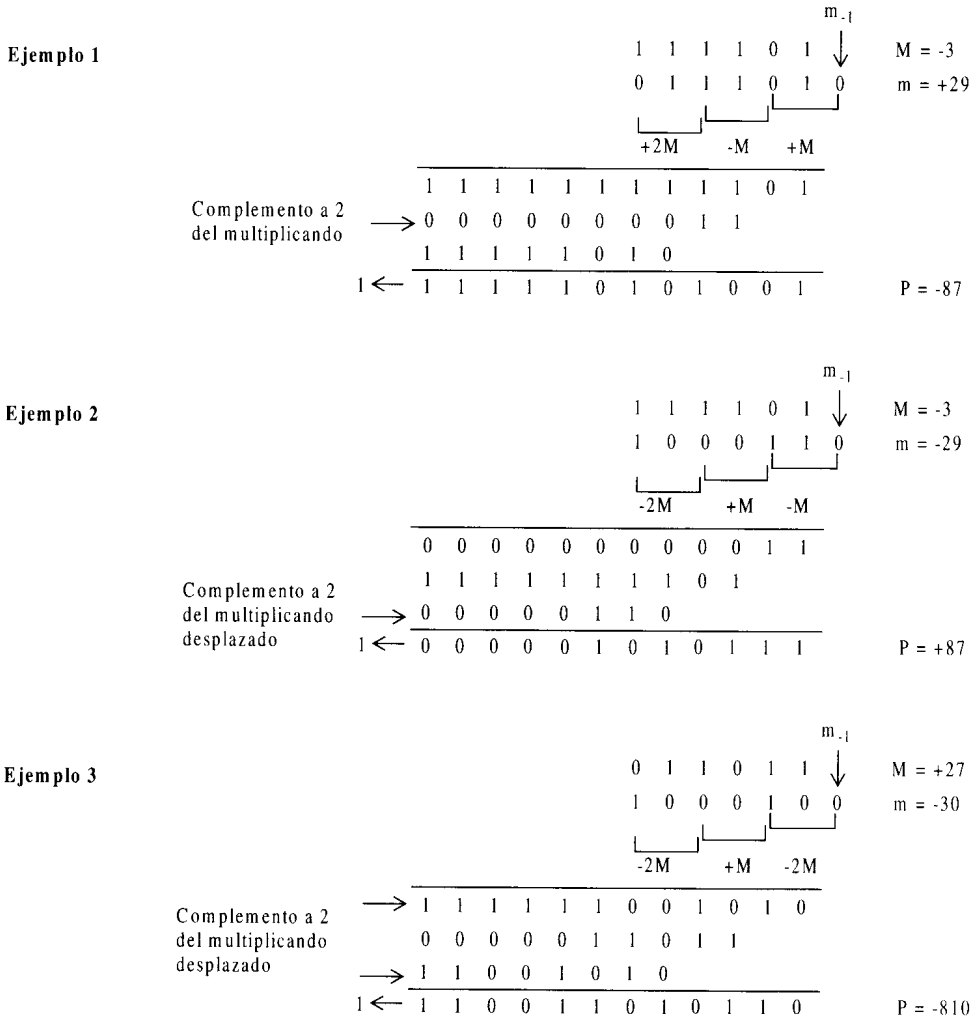


Figura 4.49: Ejemplos numéricos del algoritmo de Booth modificado

La codificación de 3 bits al tiempo, se puede extender a 4 bits cada vez, lo que reduciría el número de productos parciales de $n/2$ a $n/3$. Sin embargo, la codificación de 4 bits, requiere generar $3M$ que no es tan sencillo como generar $2M$.

Sección 4-6

Divisores binarios

Dados dos operandos, el dividendo D y el divisor d, el objetivo de los circuitos de división es calcular el cociente Q y el resto R tales que:

$$D = d \times Q + R$$

donde se requiere que el resto sea menor que el divisor, es decir $0 \leq R < d$. La expresión anterior indica una fuerte conexión entre las operaciones de multiplicar y dividir, donde el dividendo (D), el cociente (Q) y el divisor (d) se corresponden respectivamente con el producto (P), el multiplicador (m) y el multiplicando (M). Esta relación implica que los circuitos que se utilizan en ambas operaciones son análogos: en la multiplicación se suma repetidamente el multiplicando desplazado para generar el producto, y en la división el divisor desplazado se resta de forma repetida del dividendo para obtener el cociente. El método más simple de división binaria es el mismo que se realiza cuando se efectúa la división (decimal) con lápiz y papel.

4.6.1 Ejemplo: División de dos números enteros binarios sin signo

La Figura 4.50 muestra un ejemplo de división de dos números enteros binarios sin signo (39 y 6). El procedimiento que se utiliza para obtener el cociente es el siguiente:

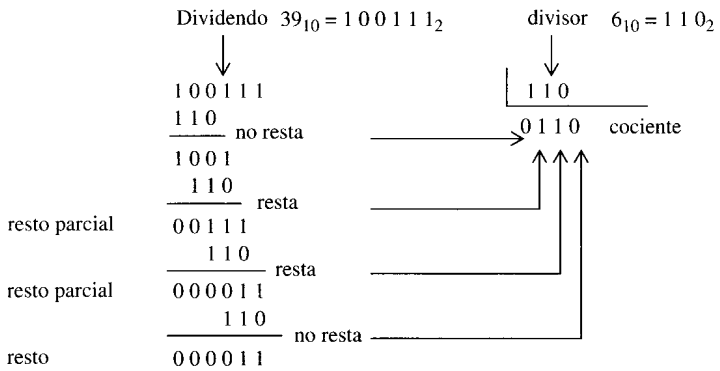


Figura 4.50: División por el método de lápiz y papel de 2 números binarios sin signo

1) Se examinan los bits del dividendo de izquierda a derecha, hasta que el conjunto de bits examinado representa un número mayor o igual que d; se dice entonces que d es capaz de *dividir* al número.

2) Hasta que ocurre este suceso se van colocando 0's en el cociente de izquierda a derecha.

3) Cuando el suceso tiene lugar, se coloca un 1 en el cociente y se resta el divisor del dividendo parcial. El resultado se conoce como *resto parcial.*

4) A partir de este punto la división sigue una estructura cíclica. En cada ciclo se añaden bits adicionales del dividendo al resto parcial, hasta que el resultado es mayor o igual que el divisor. Como antes, el divisor se resta de este número para producir un nuevo resto parcial.

5) El proceso continúa hasta que se acaban todos los bits del dividendo. ♦

La forma operativa del ejemplo presenta el inconveniente de que en el cálculo del resto parcial (R_{i+1}), el divisor que se resta de R_i sufre un desplazamiento previo hacia la derecha de un número creciente de bits:

$$R_{i+1} = R_i - Q_{n-i}\, 2^{-i} d \qquad i = 0, 1, ..., n$$

resulta más adecuado ir desplazando un lugar a la izquierda el resto parcial R_i en cada paso de la división y operar con el divisor fijo. Esto produce un efecto idéntico a desplazar a la derecha el divisor, es decir:

$$R_{i+1} = 2R_i - Q_{n-i}\, d \qquad i = 0, 1, ..., n$$

En la Figura 4.51 se muestra la misma división del ejemplo anterior realizada con este procedimiento.

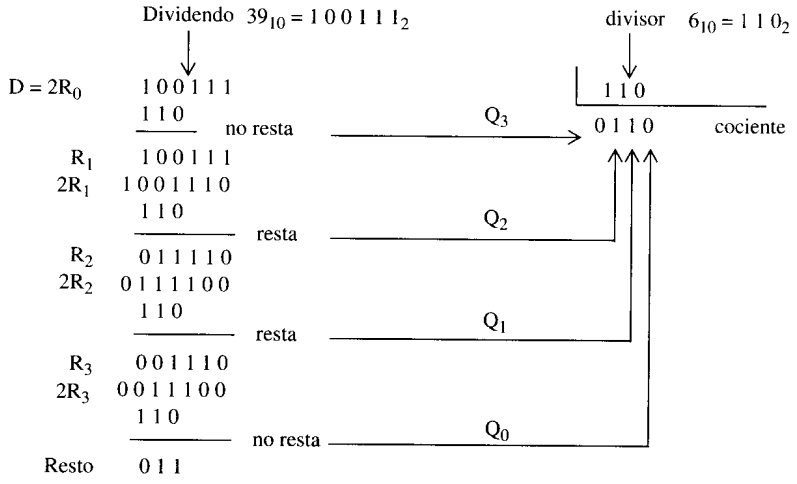


Figura 4.51: División por el método de lápiz y papel modificado

Conviene señalar que el último resto parcial R_n que se obtiene no da de forma inmediata el resto de la división, pues el resto verdadero R es: $R = R_n\, 2^{-n}$. El problema principal en la división es calcular en cada paso el valor del dígito correspondiente del cociente. Cuando se utiliza aritmética binaria este problema se aligera bastante, ya que sólo hay dos dígitos posibles para el cociente (0 ó 1). Por lo tanto, es suficiente comparar la parte más significativa de $2R_i$ con el valor de d:

$$\text{si} \quad 2R_i < d \quad \Rightarrow \quad Q_{n-i} = 0$$

$$\text{si} \quad 2R_i \geq d \quad \Rightarrow \quad Q_{n-i} = 1$$

si n es grande, la forma más práctica de efectuar la comparación anterior es hacer la resta $2R_i - d$, utilizando un sumador-restador en lugar de utilizar un circuito combinacional de comparación de números de n bits. El valor de Q_{n-i} se determina según el resultado de la operación, es decir:

$$\text{si} \quad 2R_i - d \text{ es negativo} \Rightarrow \quad Q_{n-i} = 0$$

$$\text{si} \quad 2R_i - d \text{ es positivo} \Rightarrow \quad Q_{n-i} = 1$$

En la Figura 4.52 se muestra la estructura de registros que se necesita para realizar el método de división descrito. El registro de desplazamiento de n bits Q se utiliza inicialmente para almacenar el dividendo. El divisor se carga en el registro d y permanece fijo durante toda la operación. En cada paso de la división, el conjunto registro acumulador (A) y registro cociente (Q) se desplaza un bit a la izquierda. Esto transforma el resto parcial R_i en $2R_i$. Las posiciones que van quedando vacías en la parte derecha del registro Q se pueden utilizar para ir almacenando los bits que se van generando del cociente Q_{n-i}. Cuando finaliza la división, el registro A contiene el resto de la operación y el registro Q el cociente.

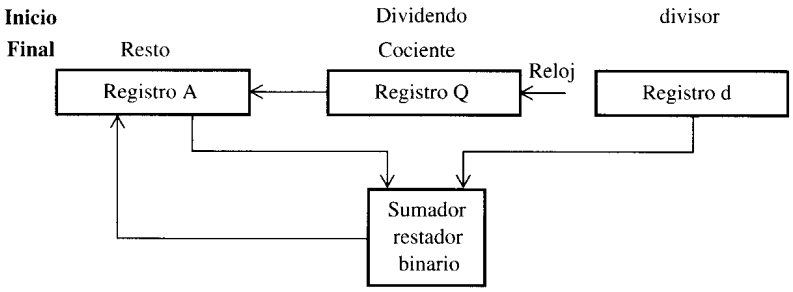


Figura 4.52: Estructura de registros para la operación de división

4.6.2 División por el método de restauración

Tal como se ha indicado, para evitar la utilización de circuitos comparadores de elevado coste, la comparación entre el dividendo y el divisor se realiza mediante una resta. En este caso el divisor se resta de la parte del dividendo que está bajo consideración.

Una respuesta positiva indica que el divisor es más pequeño, y se coloca un 1 en el cociente. Una respuesta negativa indica que el divisor es mayor, y que por lo tanto la resta no era necesaria y hay que volver a sumar el divisor al dividendo. La operación de suma *restaura* el valor original del dividendo, lo que da nombre al método. Este proceso se puede realizar sobre la estructura de registros de la Figura 4.52 mediante el algoritmo representado en la Figura 4.53.

4.6.3 Ejemplo: Algoritmo de restauración

En la Tabla 4.13 se muestra un ejemplo del algoritmo de restauración con los siguientes valores:

$$D = 11_{10} = 1011_2 \quad d = 5_{10} = 0101_2$$

se necesitan registros de 4 bits y se obtiene como resultado:

$$Q = 2_{10} = 0010_2 \qquad \text{Resto} = 1_{10} = 0001_2 \blacklozenge$$

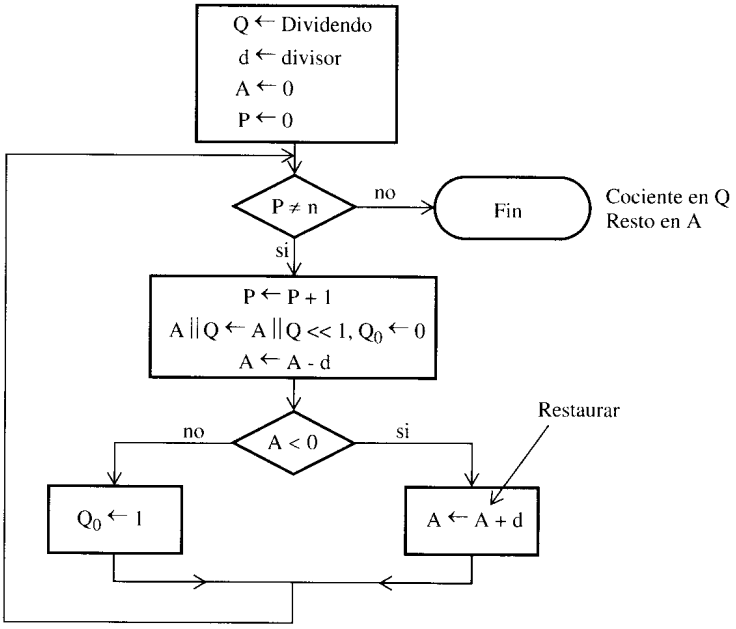


Figura 4.53: Diagrama de flujo del método de restauración

Paso	Acción	A	Q	d	P
0	Inicializar registros	0000	1011	0101	0
1	P ← P + 1 A || Q ← A || Q << 1, Q_0 ← 0 A ← A - d = A + C2(d) A < 0 ⇒ A ← A + d (Restaurar)	 0001 1100 0001	 0110 0110 0110		1
2	P ← P + 1 A || Q ← A || Q << 1, Q_0 ← 0 A ← A - d = A + C2(d) A < 0 ⇒ A ← A + d (Restaurar)	 0010 1101 0010	 1100 1100 1100		2
3	P ← P + 1 A || Q ← A || Q << 1, Q_0 ← 0 A ← A - d = A + C2(d) A ≥ 0 ⇒ Q_0 = 1	 0101 0000 0000	 1000 1000 1001		3
4	P ← P + 1 A || Q ← A || Q << 1, Q_0 ← 0 A ← A - d = A + C2(d) A < 0 ⇒ A ← A + d (Restaurar)	 0001 1100 0001 **Resto**	 0010 0010 0010 **Cociente**		4

Tabla 4.13: Ejemplo de aplicación del método de restauración

4.6.4 División por el método de no restauración

Puede evitarse el paso de restaurar del método anterior. Para ello conviene darse cuenta (ver Figura 4.53) que una restauración de la forma:

$$(R_i)_A \leftarrow (R_i)_A + d \tag{4.19}$$

va seguida en el paso siguiente por una resta

$$(R_{i+1})_A \leftarrow (2R_i)_A - d \tag{4.20}$$

donde el 2 corresponde al desplazamiento a izquierda de los registros A y Q.

$(R_i)_A$ representa que el cálculo de los sucesivos restos parciales se realiza sobre el registro A (ver Figura 4.52). Las ecuaciones (4.19) y (4.20) pueden combinarse en una sola para la obtención de R_{i+1}.

$$(R_{i+1})_A \leftarrow (2R_i)_A - d = 2((R_i)_A + d) - d \tag{4.21}$$

Esta idea es la base del método de no restauración, en el que si el bit $Q_{n-i} = 1$ el resto parcial R_{i+1} se evalúa utilizando (4.20), y si $Q_{n-i} = 0$ mediante (4.21). En la Figura 4.54 se da el algoritmo del método de no restauración.

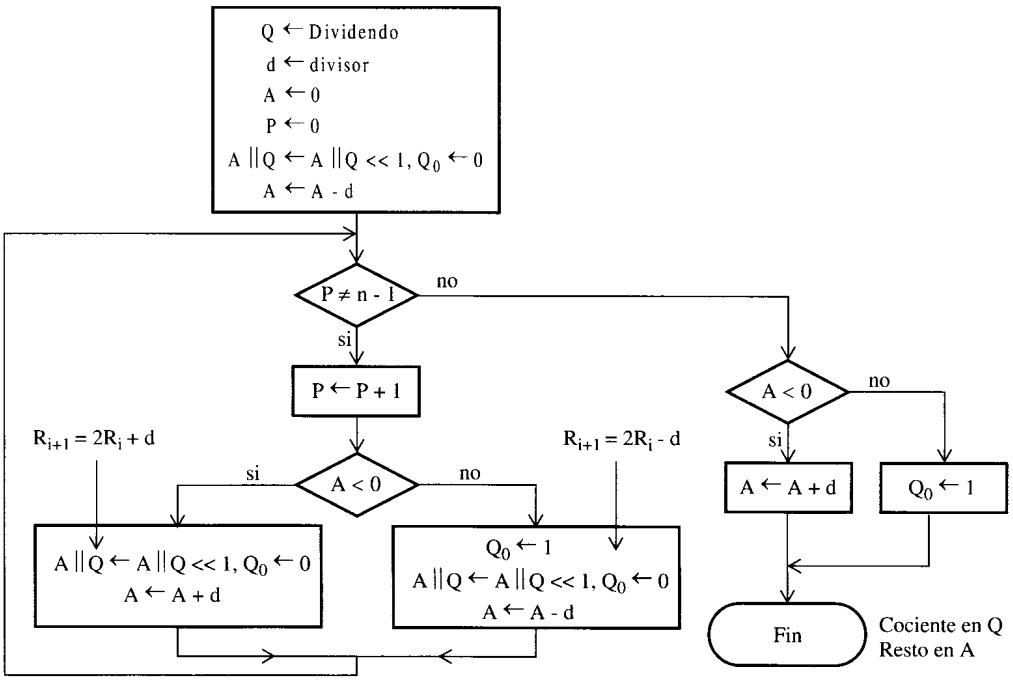


Figura 4.54: Diagrama de flujo del método de no restauración

Como los restos parciales pueden ser negativos y al final del algoritmo se desea tener un resto positivo, se modifica el último paso al salir del bucle para asegurar esta condición. Se puede ver que en este algoritmo se efectúa una suma en cada paso de la división, mientras que en el método de restauración había que realizar una suma para cada dígito del cociente que era 1 y dos sumas cuando era 0.

4.6.5 Ejemplo: Algoritmo de no restauración

En la Tabla 4.14 se muestra el mismo ejemplo que se resolvió por el método de restauración, utilizando ahora el método de no restauración. ♦

Paso	Acción	A	Q	d	P
0	Inicializar registros $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d = A + C2(d)$	0000 0001 1100	1011 0110 0110	0101	0
1	$P \leftarrow P + 1$ $A < 0$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A + d$	 1000 1101	 1100 1100		1
2	$P \leftarrow P + 1$ $A < 0$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A + d$	 1011 0000	 1000 1000		2
3	$P \leftarrow P + 1$ $A \geq 0$ $Q_0 \leftarrow 1$ $A \| Q \leftarrow A \| Q << 1, Q_0 \leftarrow 0$ $A \leftarrow A - d$	 0000 0001 1100	 1001 0010 0010		3
4	$A < 0$ $A \leftarrow A + d$	 0001 **Resto**	 0010 **Cociente**		3

Tabla 4.14: Ejemplo de aplicación del método de no restauración

Sección 4-7

Estructura de la unidad aritmético-lógica (ALU)

Los circuitos que emplea un procesador para ejecutar un programa se combinan en la unidad aritmético-lógica (ALU). La complejidad de la ALU viene impuesta de una parte, por los tipos de operaciones que puede ejecutar y, de otra, por la forma en que las efectúa. En el caso de que se empleen algoritmos análogos para la realización de las diferentes operaciones de la ALU se puede simplificar su diseño. En concreto para la multiplicación y la división es posible compartir los mismos recursos de cálculo.

La Figura 4.55 muestra la estructura de uno de los diseños de ALU más utilizado. Además de las entradas para los operandos la ALU posee diversas señales de control (sumar, restar,....), que se utilizan para seleccionar las diferentes funciones aritmético-lógicas que se desean ejecutar.

Por simplicidad, en la Figura 4.55 no se han incluido los circuitos (por otro lado inmediatos) que hacen las distintas funciones lógicas (AND, OR, NOT,...). Las operaciones de multiplicar y dividir se pueden realizar utilizando alguno de los algoritmos estudiados en los apartados anteriores a base de operaciones elementales de suma y resta.

Para el almacenamiento de los operandos y del resultado se usan 3 registros:

a) El registro acumulador (*A*)

b) El registro multiplicador/cociente (*MQ*)

c) El registro de datos (*RDAT*)

Los registros *A* y *MQ* pueden desplazarse de forma conjunta a derecha e izquierda como si fuesen un único registro. *RDAT* es un registro auxiliar empleado por la ALU.

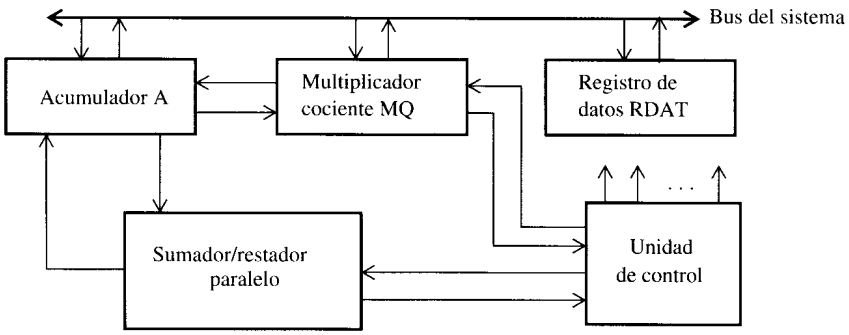


Figura 4.55: Estructura básica de una ALU

4.7.1 ALU´s integradas

Es factible empaquetar una ALU completa, del tipo que se acaba de describir, dentro de un único circuito integrado. El número de conexiones externas necesarias para controlar la ALU y poder acceder así a sus registros internos, limita la longitud de los datos que pueden tratarse por una ALU de este tipo. Es normal que dicha longitud sea de 2 ó de 4 bits. Esta anchura puede parecer pequeña, sin embargo, hay que tener presente que estas unidades se diseñan de manera que se puedan conectar fácilmente en cascada para formar una ALU capaz de procesar en paralelo un número arbitrario de bits. El objetivo de construir este tipo de módulos es doble:

1) Si se disponen de módulos aritméticos generales, es posible utilizarlos en muchas aplicaciones. Esto implica reducir el número de módulos distintos que se construyen, y aumentar así su volumen de producción con la consiguiente reducción en los costes.

2) Los módulos se pueden utilizar en procesadores, en los que la operación específica que va a ejecutar la unidad se selecciona dinámicamente por la unidad de control del procesador.

Una de las ALU's integradas que más se emplea es el circuito integrado *SN74x181* (ver Figura 4.56), que contiene las siguientes entradas:

a) 2 vectores de datos A y B de 4 bits cada uno

b) 4 señales de control (c_3, c_2, c_1, c_0) que permiten elegir una determinada función

c) 1 señal de control M para seleccionar entre funciones aritméticas y lógicas

d) 1 entrada c_e de arrastre

y como salidas:

e) 1 vector de resultado de la operación R de 4 bits

f) 1 salida c_s de arrastre

g) 2 señales P y G de propagación y generación de arrastre, que permiten construir circuitos más grandes con aceleración de arrastres

h) 1 señal EQ que indica la igualdad entre los operandos de entrada A y B

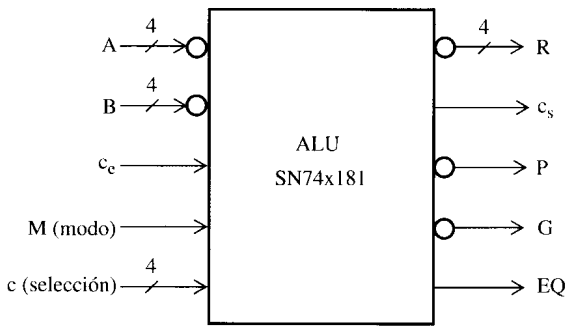


Figura 4.56: Señales de entrada-salida en la ALU SN74x181

En la Tabla 4.15 se muestra una descripción de la salida *R* del *SN74x181*. *R*, *A* y *B* representan los siguientes vectores:

$$R = (R_3, R_2, R_1, R_0)$$
$$A = (A_3, A_2, A_1, A_0)$$
$$B = (B_3, B_2, B_1, B_0)$$

v(R), v(A) y v(B) son sus respectivos valores decimales:

$$v(R) = \sum_{i=0}^{3} R_i 2^i \qquad v(A) = \sum_{i=0}^{3} A_i 2^i \qquad v(B) = \sum_{i=0}^{3} B_i 2^i$$

Selección $c_3c_2c_1c_0$	**M = 1 Función lógica**	**M = 0 Función aritmética v(R) = s mod 16**
0 0 0 0	$R = \overline{A}$	$s = v(A) - 1 + c_e$
0 0 0 1	$R = \overline{A} \vee \overline{B}$	$s = v(A \wedge B) - 1 + c_e$
0 0 1 0	$R = \overline{A} \vee B$	$s = v(A \wedge \overline{B}) - 1 + c_e$
0 0 1 1	R = 1 1 1 1	$s = 1\,1\,1\,1 + c_e$
0 1 0 0	$R = \overline{A} \wedge \overline{B}$	$s = v(A) + v(A \vee \overline{B}) + c_e$
0 1 0 1	$R = \overline{B}$	$s = v(A \wedge B) + v(A \vee \overline{B}) + c_e$
0 1 1 0	$R = A \oplus \overline{B}$	$s = v(A) - v(B) - 1 + c_e$
0 1 1 1	$R = A \vee \overline{B}$	$s = v(A \vee \overline{B}) + c_e$
1 0 0 0	$R = \overline{A} \wedge B$	$s = v(A) + v(A \vee B) + c_e$
1 0 0 1	$R = A \oplus B$	$s = v(A) + v(B) + c_e$
1 0 1 0	R = B	$s = v(A \wedge \overline{B}) + v(A \vee B) + c_e$
1 0 1 1	$R = A \vee B$	$s = v(A \vee B) + c_e$
1 1 0 0	R = 0 0 0 0	$s = v(A) + v(A) + c_e$
1 1 0 1	$R = A \wedge \overline{B}$	$s = v(A \wedge B) + v(A) + c_e$
1 1 1 0	$R = A \wedge B$	$s = v(A \wedge \overline{B}) + v(A) + c_e$
1 1 1 1	R = A	$s = v(A) + c_e$

Tabla 4.15: Funciones de la ALU SN74181

El significado de los operadores que aparecen en la Tabla 4.15 es el siguiente:

$A \wedge B$ AND de los vectores A y B

$A \vee B$ OR de los vectores A y B

v(A) + v(B)	suma aritmética de los vectores A y B
v(A) - v(B)	resta aritmética de los vectores A y B
$\overline{A}$, $\overline{B}$	NOT (A), NOT (B)

Las otras señales de salida vienen descritas por:

$c_s = 1$ si $s \geq 16$ ó $s < 0$

$c_s = 0$ en cualquier otro caso

$EQ = R_3 \wedge R_2 \wedge R_1 \wedge R_0$ detecta la igualdad de A y B

La entrada M del *SN74x181* selecciona entre operaciones aritméticas y lógicas. Cuando $M = 1$, se seleccionan las operaciones lógicas y cada salida R_i es función sólo de los correspondientes datos de entrada A_i y B_i. Ningún arrastre se propaga entre etapas y se ignora la entrada c_e. Las entradas c_3, c_2, c_1 y c_0 determinan una operación lógica particular. Se puede seleccionar cualquiera de las 16 funciones lógicas combinacionales de dos variables que hay.

Cuando $M = 0$, se seleccionan operaciones aritméticas, se propagan los arrastres entre las etapas y c_e se utiliza como un arrastre de entrada a la etapa menos significativa. Para aquellas operaciones que requieran más de 4 bits, se pueden disponer en cascada múltiples ALUS's *SN74x181* de la misma forma que se conectaban cuatro sumadores con aceleración de arrastres de 4 bits (ver Figura 4.18) para construir un sumador de 16 bits. El arrastre de salida c_s de cada ALU se conecta al arrastre de entrada c_e de la siguiente etapa más significativa. A todas las ALU's se les aplican las mismas señales (M, c_3, c_2, c_1 y c_0) para seleccionar la función que realiza.

Por ejemplo para efectuar la *suma en complemento a 2*, con las entradas c_3, c_2, c_1 y c_0 se selecciona la función v(A) + v(B) + c_e. La entrada c_e a la ALU menos significativa normalmente se pone a 0 durante la operación de suma. Si la operación fuese la *resta en complemento a 2*, con las entradas c_3, c_2, c_1 y c_0 se selecciona la función v(A) - v(B) - 1 + c_e. En este caso la entrada c_e a la ALU menos significativa normalmente se pone a 1, porque c_e actúa como el complemento del dígito que se lleva durante la operación de resta. El *SN74x181* proporciona otras operaciones aritméticas, tales como v(A) - 1 + c_e que es útil en algunas aplicaciones (por ejemplo decrementar en 1). Dispone también de un conjunto de operaciones aritméticas un tanto esotéricas como v $(A \wedge \overline{B})$ + v $(A \vee B)$ + c_e, que casi nunca se emplean en la práctica.

Debe observarse (ver Figura 4.56) que las entradas de los operandos A_3, A_2, A_1, A_0, B_3, B_2, B_1, B_0 y las salidas R_3, R_2, R_1, R_0 del *SN74x181* son activas en baja. El *SN74x181* se puede utilizar con operandos y salidas que son activas en alta. En este caso se debe construir una versión diferente de la Tabla 4.15. Cuando $M = 1$, las funciones lógicas que se obtienen son precisamente las duales de las que se muestran en la Tabla 4.15. Cuando $M = 0$, las funciones aritméticas son distintas de las que se dan en la Tabla 4.15.

Sección 4-8

Aritmética en coma flotante

La notación en *coma fija* resulta conveniente para representar valores enteros de magnitud pequeña y números fraccionarios escalados, es decir, con la posición de la coma decimal fijada a priori. Todos los algoritmos de las operaciones aritméticas que se han visto hasta ahora suponen que los números están representados en coma fija.

Estos mismos algoritmos se podrían aplicar también a los números reales representados en *coma flotante* a condición de tener en cuenta el escalado, lo que implica mantener en todo momento un conocimiento de la posición correcta donde se localiza la separación de las partes entera y decimal. Es posible manejar los problemas de escalado mediante programación, sin embargo el programa resultante tiende a ser ineficaz debido a los pasos adicionales que se requieren para mantener los factores de escala, e impone al programador una tarea innecesaria y propensa a errores.

Los procesadores en coma flotante manejan los factores de escala de forma automática. El hardware es más complejo, pero como contrapartida tiene la ventaja de que la operación del computador es mucho más eficaz. Estas consideraciones son las que justifican el diseño de una unidad aritmética en coma flotante.

Para poner de manifiesto la necesidad de la representación de números en coma flotante, conviene en primer lugar recordar como maneja un computador los números en coma fija. La forma más natural en principio para operar con números grandes en una máquina que posee una longitud de palabra pequeña es encadenar varias de estas palabras para disponer de un mayor rango para representar los números. Una primera cuestión que surge es la siguiente: ¿cómo trata el computador la parte decimal de un número?. En la representación en coma fija, afortunadamente esto no plantea ningún problema. A modo ilustrativo considérense los dos cálculos siguientes utilizando aritmética decimal.

Caso 1: Aritmética entera

$$\begin{array}{r} 7\,6\,3\,2\,1\,3\,5 \\ +\ 1\,7\,9\,4\,8\,2\,1 \\ \hline 9\,4\,2\,6\,9\,5\,6 \end{array}$$

Caso 2: Aritmética en coma fija

$$\begin{array}{r} 7\,6\,3{,}2\,1\,3\,5 \\ +\ 1\,7\,9{,}4\,8\,2\,1 \\ \hline 9\,4\,2{,}6\,9\,5\,6 \end{array}$$

Aunque el primer caso utiliza *aritmética entera* y el segundo *aritmética decimal* (es decir con parte fraccionaria) los cálculos son totalmente idénticos. Este principio se puede extender a la unidad aritmética de un computador. Todo lo que un programador tiene que hacer es recordar donde está colocada la coma. Todas las entradas al computador se escalan de forma que se adapten a este convenio y todas las salidas están también escaladas de forma análoga. Las operaciones internas se realizan como si los números fuesen enteros. Esta organización es lo que se conoce como aritmética en coma fija, porque la coma binaria (que

separa la parte entera de la fraccionaria) se supone que permanece en la misma posición. Es decir hay siempre el mismo número de dígitos antes y después de la coma. La ventaja de la representación de números en coma fija es que para su implementación no se necesita un software o un hardware especialmente complejo. Un sencillo ejemplo pone de manifiesto esta idea de su sencillez.

Sea un número de 8 bits, donde los 4 más significativos representan la parte entera y los 4 menos significativos la parte fraccionaria. Se desea imprimir la suma de los dos números siguientes: 3,625 y 6,5. Como primer paso el programa de entrada convierte ambos números a su forma binaria:

$$3{,}625_{10} \rightarrow 11{,}101_2 \rightarrow 0011{,}1010_2 \text{ (en 8 bits)}$$

$$6{,}5_{10} \rightarrow 110{,}1_2 \rightarrow 0110{,}1000_2 \text{ (en 8 bits)}$$

El computador ahora considera estos números como 00111010 y 01101000 respectivamente. Conviene recordar que a todos los efectos la coma decimal es imaginaria. Estos números se suman de la forma usual y se obtiene:

```
  0 0 1 1 1 0 1 0
  0 1 1 0 1 0 0 0
-----------------
1 1 0 1 0 0 0 1 0 = 162 (si se interpreta como un número binario sin signo de 8 bits)
```

El programa de salida ahora toma el resultado y lo separa en su parte enterra 10100, y en su parte fraccionaria, 0010 e imprime la respuesta correcta 10,125. Conviene observar que un número representado en coma fija puede utilizar algunas palabras para conseguir un mayor rango de valores que el que permite una palabra simple. Sin embargo los números en coma fija tienen sus limitaciones. Considérese, por ejemplo el trabajo de un astrofísico que investiga la conducta del sol. Puede verse confrontado con cantidades tales como la masa del sol (1 990 000 000 000 000 000 000 000 000 000 000 g) y la masa de un electrón 0,000 000 000 000 000 000 000 000 910 956 g).

Si el astrofísico emplea para sus cálculos aritmética en coma fija, necesitaría un número extraordinariamente grande de bytes para representar todo el rango de números. Un único byte representa números en el rango 0-255 o aproximadamente de 0 a 1/4 de millar. Si el astrofísico necesitase trabajar simultáneamente con números astronómicamente grandes y microscópicamente pequeños, requeriría unos 14 bytes para la parte entera y 12 bytes para la parte fraccionaria, es decir ¡un número de 208 bits!. Una pista para resolver el problema del gran número de bits está en el hecho de que ambos números contienen un gran número de ceros y pocos dígitos significativos.

4.8.1 La representación de números en coma flotante

Los computadores a menudo manejan, representan, y almacenan números en un formato de *coma flotante*. De la misma forma que el número decimal 124,56 se puede representar como $0{,}12456 \times 10^3$, un computador maneja los números binarios de una forma similar. Por ejemplo, 1101101,1101101 se puede representar internamente como $0{,}11011011101101 \times 2^7$ (el 7 se almacena también en formato binario). La notación en coma flotante se denomina algunas veces *notación científica*. Antes de considerar más detalladamente los números en coma flotante es necesario considerar las ideas de *rango, precisión y exactitud* que están estrechamente relacionadas con la forma en que los números se representan en coma flotante.

- **Rango**. El rango de un número indica cuán grande o cuán pequeño puede ser. En el ejemplo del astrofísico se consideraban números tan grandes como 2×10^{23} y tan pequeños como 9×10^{-28},

lo que representa un rango de 10^{51} o 51 décadas. El rango de los números representados en un computador debe ser suficiente para la gran mayoría de los cálculos que son probables que se puedan efectuar. Si se va a utilizar el computador en una aplicación específica donde se conoce a priori que el rango de los datos es pequeño entonces el rango de los números válidos puede restringirse, simplificando los requisitos del sistema.

- **Precisión**. La precisión de un número se corresponde con el número de cifras significativas utilizadas para representarlo. Por ejemplo la constante π se puede escribir como 3,142 ó 3,141592. El último caso es más preciso que el primero porque representa a π en una parte en 10^7 mientras el primero lo hace en una parte en 10^4.
- **Exactitud**. La precisión es una medida de la fidelidad de una cantidad. Por ejemplo, se puede decir que $\pi = 3{,}141$ ó $\pi = 3{,}241592$. Se puede ver que en el primer caso se tiene un número que tiene poca precisión en comparación con el primero pero que sin embargo es más exacto. En un mundo ideal, precisión y exactitud deberían ir de la mano. Sin embargo en el mundo real con computadores que poseen longitudes de palabra finita se está confrontado con el diseño de algoritmos numéricos que garanticen la exactitud que la precisión disponible permite.

Un número *x* en coma flotante se representa en la forma siguiente:

$$x = m \times B^e$$

donde *m* recibe el nombre de *mantisa* o *argumento*, *e* es el *exponente* y *B* la *base o raíz*. La forma en que un computador almacena los números en coma flotante es dividiendo la secuencia binaria que lo representa en dos campos tal como se ilustra en la Figura 4.57.

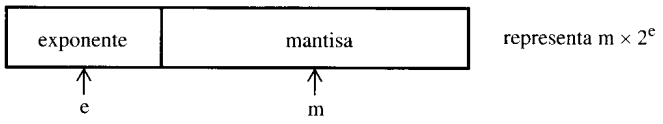


Figura 4.57: Almacenamiento de un número binario en coma flotante como exponente y mantisa

La base *B* está implícita y no necesita almacenarse, puesto que es la misma para todos los números. En lo sucesivo se supondrá que la base es dos. No es necesario que un número en coma flotante ocupe una única posición de memoria. Si la longitud de palabra es de 8 bits este tipo de representación no sería de utilidad. Con frecuencia una serie de palabras se agrupan para formar el número en coma flotante (ver Figura 4.58). La separación entre exponente y mantisa no necesita caer en la frontera de una palabra. Esto es, una mantisa podría típicamente ocupar tres bytes y el exponente un byte en un número en coma flotante que ocupa dos palabras de 16 bits.

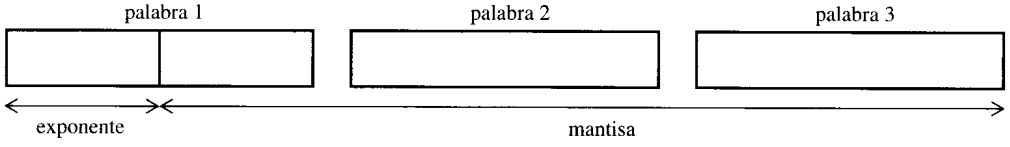


Figura 4.58: Utilización de algunas palabras para representar un número en coma flotante

4.8.2 La normalización de números en coma flotante

Por convenio la mantisa en coma flotante se *normaliza* siempre (a menos que sea igual a cero) de manera que se expresa en la forma $0,1.....\times 2^e$. Por el momento sólo se consideran mantisas positivas. Si el resultado de un cálculo fuera $0,01.....\times 2^e$ se normalizaría para dar $0,1.....\times 2^{e-1}$. Análogamente, el resultado $1,01.....\times 2^e$ se normalizaría a $0,101.....\times 2^{e+1}$. Una de las ventajas que proporciona normalizar la mantisa es que se optimiza la precisión con la que se opera en los cálculos. Por ejemplo, la mantisa de 8 bits no normalizados 0,00001010 tiene solamente 4 bits significativos, mientras que su valor normalizado 0,10100011 tiene 8 bits significativos. Existe sin embargo una ligera diferencia entre los números decimales normalizados tal como se utilizan en ciencias e ingenierías y los números binarios normalizados. Un número decimal en coma flotante se normaliza de forma que su mantisa está en el rango [1,00....0 9,99....9]. La mantisa normalizada *m* de un número binario positivo en coma flotante es de la forma:

$$m \in [0,100...0 \quad 0,111...1]$$

Es decir $1/2 \leq m < 1$. Una excepción especial tiene que hacerse con el caso del cero, ya que este número no se puede normalizar. En la representación en complemento a 2 la mantisa normalizada *m* de un número binario negativo en coma flotante se almacena de la forma:

$$m \in [1,011...1 \quad 1,000...0]$$

En este caso la mantisa negativa *m* pertenece al intervalo $-1/2 > m \geq -1$. Se ve pues que los números en coma flotante están restringidos a uno de los tres rangos descritos en la Figura 4.59.

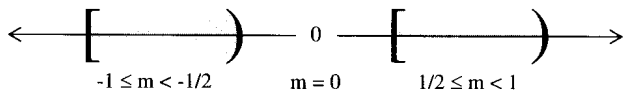


Figura 4.59: Rango de mantisas normalizadas en complemento a 2 que son válidas

En la representación magnitud-signo la mantisa normalizada de un número binario en coma flotante está en el rango [0,100...0 0,111...1] para números positivos y [1,100...0 1,111...1] para números negativos. Como se verá posteriormente, el formato IEEE para números en coma flotante emplea una representación del tipo magnitud-signo con un 1 a la izquierda de la coma (las mantisas caen en el rango [-1,11...1 -1.00...0] ó [1.00...0 1,11...1]. La mantisa *m* está ligada a pertenecer a los intervalos $-2 < m \leq 1$, $m = 0$ ó $1 \leq m < 2$.

4.8.3 Exponentes polarizados

Una representación en coma flotante de números debe considerar mantisas y exponentes tanto positivos como negativos. Por ejemplo en notación decimal esto corresponde a:

$$+0,123 \times 10^{12} \qquad -0,756 \times 10^{9}$$

$$+0,176 \times 10^{-3} \qquad -0,459 \times 10^{-7}$$

La mantisa de un número en coma flotante se representa a menudo como un número en complemento a 2. Sin embargo, el exponente a veces se representa en forma *polarizada*. Si se tiene un exponente de *m* bits, hay 2^m posibles números enteros sin signo desde 000...0 a 111...1. Supóngase ahora que estos números se

reetiquetan no desde 0 hasta 2^m-1 sino desde -2^{m-1} a $+2^{m-1}$-1 al restar un valor constante (o polarización) de 2^{m-1} de cada uno de los números. Lo que se tiene así es una serie binaria natural continua desde 0 a N que representa a los números desde P a N - P. Por ejemplo, si la serie (en decimal) fuera 0, 1, 2, 3, 4, 5, 6, 7 se podría restar P = 4 de cada número de la serie para generar una nueva serie -4, -3, -2, -1, 0, 1, 2, 3. Esto es realmente un nuevo método de representar números negativos añadiendo una constante al número más negativo para que el resultado sea igual a cero. En el ejemplo anterior, se ha añadido 4 a cada número de forma que -4 se representa por 0, -3 por +1 etc.

Se crea así un exponente polarizado sumando una constante al exponente verdadero de forma que el exponente polarizado viene dado por $e' = e + P$, donde e' es el exponente polarizado, e es el exponente verdadero y P un sumando de ponderación. El valor de P es con frecuencia 2^{m-1} ó 2^{m-1}-1. En la Tabla 4.16 se muestra lo que sucede en el caso en que m = 4 y $P = 2^3 = 8$. Por ejemplo, si x = 1010,1111 se normaliza a 0,10101111 $\times$ 2^4. El exponente verdadero es e = +4 que se almacena como un exponente polarizado $e' = 4 + 8 = 12 = 1100_2$. La representación polarizada del exponente presenta algunas ventajas:

Representación binaria	Exponente verdadero	Forma polarizada
0 0 0 0	-8	0
0 0 0 1	-7	1
0 0 1 0	-6	2
0 0 1 1	-5	3
0 1 0 0	-4	4
0 1 0 1	-3	5
0 1 1 0	-2	6
0 1 1 1	-1	7
1 0 0 0	0	8
1 0 0 1	1	9
1 0 1 0	2	10
1 0 1 1	3	11
1 1 0 0	4	12
1 1 0 1	5	13
1 1 1 0	6	14
1 1 1 1	7	15

Tabla 4.16: Relación entre los exponentes verdadero y polarizado (P = 8)

1) El exponente más negativo se representa por cero. Por convenio el valor en coma flotante del cero en el sistema de exponente polarizado (ver Figura 4.60) se representa por 0,00....0 $\times$ 2^0 que es una mantisa y un exponente igual a cero (que es el exponente más negativo). En realidad el valor 0 se podría representar con m = 0 y cualquier exponente porque evidentemente 0 $\times$ 2^n = 0.

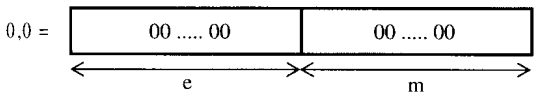


Figura 4.60: Representación del cero con un exponente polarizado

Sin embargo, puede ocurrir que al realizar determinadas operaciones que deben dar un resultado de 0, a causa de los errores de redondeo, se genere algún 1 en las posiciones menos significativos de la mantisa. Esto sugiere que el exponente más adecuado para representar el 0 sea precisamente aquél que tiene el mayor valor negativo. De esta manera se logra que los errores de redondeo producidos en la mantisa, den un número tan próximo a cero como sea posible.

2) La representación del cero es una secuencia de 0's tanto en la mantisa como en el exponente. Se evitan así ambigüedades en la representación del cero en coma flotante y se simplifica la comprobación de la igualdad de un número con 0. Teóricamente pueden existir muchas maneras de representar un cero en coma flotante (siempre que la mantisa sea cero y con cualquier valor del exponente). En algunos computadores, cuando un cálculo produce una mantisa igual a cero, el exponente se deja en el valor que tenía al final de la operación, lo que provoca que el cero no sea único. Una característica deseable, en el diseño de cualquier procesador, es tener una única representación para el cero. En aritmética en coma fija, el cero es único y se representa por un número en el que todos sus bits son ceros.

3) Cuando se suman o restan dos números en coma flotante se deben comparar los exponentes y hacerlos iguales, lo que resulta en una operación de *desplazamiento o alineamiento* para una de las mantisas. La comparación de los exponentes es relativamente directa, ya que los exponentes son siempre números positivos. Es suficiente un simple comparador.

4) Los exponentes que se almacenan forman una secuencia binaria natural. Esta secuencia es monótona creciente de forma que si se aumenta (disminuye) el exponente en 1 supone sumar (restar) 1 al exponente binario. En ambos casos el exponente binario polarizado se puede considerar que se comporta como un número binario sin signo.

4.8.4 Posibles sistemas en coma flotante

Para definir una representación en coma flotante para un determinado computador hay que seleccionar los siguientes elementos:

1) El número de palabras utilizadas
2) La representación de la mantisa (complemento a 2, magnitud-signo, etc)
3) La representación del exponente (polarizado, no polarizado, etc)
4) El número de bits dedicados a la mantisa y al exponente
5) La localización de la mantisa (antes o después del exponente)

El punto 4) merece algún comentario adicional. Una vez que se ha decidido el número total de bits en la representación en coma flotante (un número entero de palabras) hay que hacer una partición entre la mantisa y el exponente. Si se dedica un gran número de bits al exponente, el resultado es un número en coma flotante con un rango muy grande. Estos bits del exponente se han obtenido a expensas de la mantisa lo que reduce la precisión del número en coma flotante. Inversamente, si se aumentan los bits que están disponibles para la mantisa se mejora la precisión a costa del rango.

Debido a los cinco puntos anteriores, hay una multitud de formatos para la representación de números en coma flotante. Los ejemplos de la Figura 4.61 ilustran la representación de números en coma flotante

adoptada por algunos computadores (prácticamente cada computador utilizaba un formato diferente del resto). Con la aparición del microprocesador y la especificación introducida por el IEEE para números en coma flotante esta situación ha cambiado considerablemente.

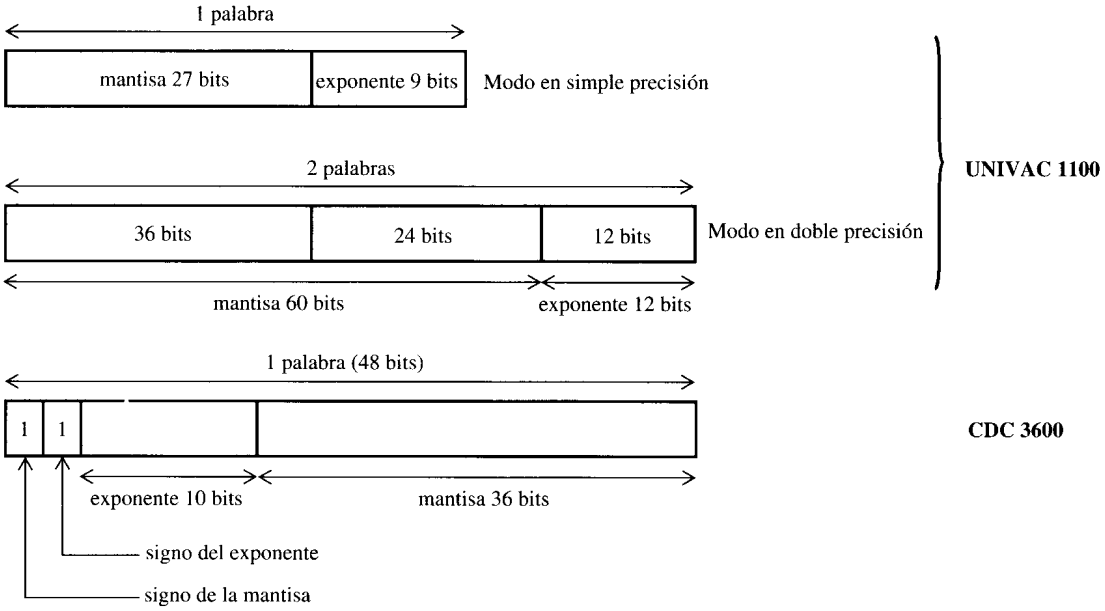


Figura 4.61: Algunos formatos de representación de números en coma flotante

El modo en simple precisión de Univac proporciona un rango de aproximadamente 10^{-76} a 10^{+76} con una precisión de 8 cifras decimales. En el modo en doble precisión el rango se aumenta hasta 10^{-614} a 10^{+614} y la precisión es equivalente a 18 cifras decimales. El modo en doble precisión se utiliza en aquellos cálculos numéricos donde se requiere una gran precisión. En general su empleo hace considerablemente más lenta la ejecución de los programas a menos que el computador posea una unidad en coma flotante especial de alta velocidad.

4.8.5 El formato IEEE de representación de números en coma flotante

El Institute of Electrical and Electronics Engineers (IEEE) ha propuesto un estándar de representación de números en coma flotante para las operaciones aritméticas en mini y microcomputadores (norma ANSI/ IEEE 754-1985). Con el fin de permitir su utilización en diferentes aplicaciones, IEEE especifica tres formatos básicos, llamados *simple*, *doble* y *cuádruple*. En la Tabla 4.17 se definen las características principales de estos tres formatos de representación de números en coma flotante. Los números se normalizan de forma que sus mantisas están contenidas en el rango $1 \leq m < 2$. Este rango corresponde a una mantisa con una parte entera igual a 1. Un número en coma flotante en el formato IEEE se define formalmente como:

$$x = -1^{s} \times 2^{e-P} \times 1{,}m$$

donde:

s = bit de signo, 0 = mantisa positiva, 1 = mantisa negativa

e = exponente polarizado por P

m = parte fraccionaria de la mantisa (la mantisa es 1,m donde el 1 de delante está implícito)

Tipo	Simple	Doble	Cuádruple
Anchura del campo en bits			
s = signo	8	11	15
e = exponente	1	1	1
d = bit delante de la mantisa	1	1	1
m = fracción de la mantisa	23	52	111
Anchura total	32	64	128
Bit de signo	0 = +, 1 = -	0 = +, 1 = -	0 = +, 1 = -
Exponente			
e máximo	255	2047	32767
e mínimo	0	0	0
polarización	127	1023	16383

Tabla 4.17: Formatos IEEE básicos en coma flotante

Así por ejemplo, un número en coma flotante de 32 bits en formato simple tiene una polarización de 127 y una parte fraccionaria para la mantisa de 23 bits. Hay dos puntos de particular interés que conviene precisar. El primero es que el formato IEEE adopta una representación magnitud-signo para la mantisa. Si $s = 1$ la mantisa es negativa y si $s = 0$ es positiva. El segundo es que la mantisa está siempre normalizada y pertenece al rango [1,000...000, 1,111...111]. Se observa que la normalización de un número en el formato IEEE es diferente a lo que se había comentado anteriormente. Si la mantisa está siempre normalizada, entonces el 1 de delante de la coma (la parte entera) es redundante cuando un número en el formato IEEE se almacena en memoria. Si se sabe que hay un 1 a la izquierda de la parte fraccionaria de la mantisa, no hay necesidad de almacenarlo (por esta causa a veces se le denomina *bit escondido*). De esta manera se elimina un bit en el almacenamiento, lo que permite mejorar la precisión de la mantisa al ampliarla en un bit. En la Figura 4.62 se muestra el formato de un número cuando se almacena en memoria.

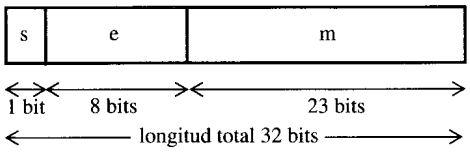


Figura 4.62: Formato IEEE del tipo simple de 32 bits

4.8.6 Ejemplo: Representación en el formato IEEE

Como ejemplo de utilización del formato simple de 32 bits de IEEE se considera la representación del número decimal -2345,125 en un computador que posee una longitud de palabra de 16 bits.

$$-2345{,}125_{10} = -100100101001{,}001_2 \text{ (número binario equivalente)}$$

$$= -1{,}00100101001001 \times 2^{11} \text{ (número binario normalizado)}$$

Para especificar completamente el número hay que determinar el bit de signo s, el exponente polarizado e y la parte fraccionaria de la mantisa m.

1) La mantisa es negativa así que el bit de signo es $s = 1$

2) El exponente polarizado viene dado por $+11 + 127 = 138 = 10001010_2$

3) La parte fraccionaria de la mantisa es ,00100101001001000000000 (en 23 bits)

Este número se almacena en dos palabras consecutivas de 16 bits:

Con el fin de minimizar el espacio de almacenamiento en una memoria de 16 bits, los números en coma flotante se *empaquetan* de forma que el bit de signo, el exponente y la mantisa comparten parte de dos o más palabras de memoria. Cuando se va a realizar la operación en coma flotante, en primer lugar lo que se hace es desempaquetar los números y se separa la mantisa del exponente. Por ejemplo, el formato básico en simple precisión especifica una mantisa con una parte fraccionaria de 23 bits, que produce una mantisa de 24 bits cuando se desempaqueta y se reinserta el 1 que va por delante. Si el procesador en el que se van a procesar los números en coma flotante tiene una longitud de palabra de 16 bits, la mantisa desempaquetada ocupará 24 bits de los 32 bits que se disponen en las dos palabras.

Si cuando se desempaqueta un número, se permite que aumente el número de bits en su exponente y en su mantisa para rellenar todo el espacio disponible se dice que el formato es *extendido*. Al extender el formato de esta forma, se incrementa considerablemente el rango y la precisión del número en coma flotante. Por ejemplo, un número en formato simple se almacena en 32 bits. Cuando se desempaqueta la parte fraccionaria de la mantisa de 23 bits pasa a 24 bits al incluir el 1 que va por delante y la mantisa se amplía a 32 bits (como una palabra simple de 32 bits o como dos palabras de 16 bits).

Todos los cálculos se hacen con la precisión de los 32 bits de la mantisa extendida. Esto es particularmente ventajoso cuando se evaluan funciones trascendentes (p. ej. sen (x) ó cos (x)). Después de efectuar, en el formato extendido, una secuencia de operaciones el número en coma flotante se reempaqueta y almacena en memoria en su formato básico. En el formato simple de 32 bits, el exponente máximo e_{max} es +127 y el exponente mínimo e_{min} es -126 y no +128 y -127 como se podría esperar. El valor especial $e_{min} - 1$ (-127) se utiliza para codificar el cero y $e_{max} + 1$ para codificar más o menos infinito o una condición que se designa como *NaN* (Not a Number). Un *NaN* es una entidad especial que incorpora el formato IEEE y que permite la manipulación de formatos fuera del estándar.

4.8.7 Operaciones aritméticas en coma flotante

Dados los números:

$$x = m_x 2^{e_x} \qquad y = m_y 2^{e_y}$$

la Tabla 4.18 resume las operaciones básicas de la arimética en coma flotante (se supone $e_x \leq e_y$).

x + y	$(m_x 2^{e_x - e_y} + m_y) 2^{e_y}$
x - y	$(m_x 2^{e_x - e_y} - m_y) 2^{e_y}$
x × y	$(m_x \times m_y) 2^{e_x + e_y}$
x ÷ y	$(m_x \div m_y) 2^{e_x - e_y}$

Tabla 4.18: Operaciones aritméticas en coma flotante

En la Tabla 4.18 se observa que la mantisa del exponente más pequeño se desplaza e_x - e_y lugares a la derecha para la suma y la resta. Todas las operaciones aritméticas en coma flotante pueden producir una condición de *rebose*, si el resultado es demasiado grande (desbordamiento) o demasiado pequeño (subdesbordamiento), para poder representarse en la máquina. Los tipos de rebose a considerar son:

1) *Desbordamiento del exponente.* Un exponente positivo e, excede su valor máximo permitido. En algunos sistemas, el número x puede representarse como $+\infty$ ó $-\infty$.

2) *Subdesbordamiento del exponente.* Un exponente negativo e, excede su valor máximo posible. Esto significa que el número x es demasiado pequeño para poderse representar y se le puede considerar igual a 0.

3) *Subdesbordamiento de la mantisa.* Se puede producir en el proceso de alineamiento de las mantisas, si los dígitos se desplazan hacia la derecha más allá de su bit menos significativo. Cuando sucede esto se precisa alguna forma de redondear el resultado.

4) *Desbordamiento de la mantisa.* La suma de dos mantisas del mismo signo puede generar un arrastre del bit más significativo. Esto se puede corregir mediante una operación de renormalización que se explica posteriormente (desplazando un bit a la derecha la mantisa y ajustando el exponente).

4.8.8 Algoritmo de suma y resta en coma flotante

Al contrario de lo que pasa con los números enteros o en coma fija, los números en coma flotante no se pueden sumar en una única operación. Para poner esto de manifiesto considérese el siguiente ejemplo utilizando aritmética decimal. Sea x = 12345 e y = 567,89. En coma flotante estos dos números se pueden representar por:

$$x = 0{,}12345 \times 10^5 \qquad y = 0{,}56789 \times 10^3$$

Si estos dos números se fueran a sumar de forma manual no surgiría ningún problema:

$$\begin{array}{r} 12345 \\ +\quad 567{,}89 \\ \hline 12912{,}89 \end{array}$$

Sin embargo, cuando se expresan en formato normalizado se plantea el problema siguiente:

$$\begin{array}{r} 0{,}1\,2\,3\,4\,5 \times 10^5 \\ + \underline{0{,}5\,6\,7\,8\,9 \times 10^3} \end{array}$$

no puede realizarse mientras los exponentes sean *diferentes*. Para efectuar una suma (o una resta) en coma flotante hay que realizar los siguientes pasos:

1) Seleccionar el número con menor exponente y desplazar su mantisa a la derecha, un número de pasos igual a la diferencia en valor absoluto de los exponentes de los dos operandos.
2) Hacer el exponente del resultado igual al mayor de los exponentes de los operandos.
3) Realizar la suma (resta) de las mantisas y determinar el signo del resultado.
4) Normalizar el resultado si es necesario.
5) Comprobar condiciones de rebose.

En el ejemplo anterior se tenía $x = 0{,}12345 \times 10^5$ e $y = 0{,}56789 \times 10^3$. El exponente de y es más pequeño que el de x lo que resulta en un incremento de 2 en su exponente y en la correspondiente división de su mantisa por 10^2 para obtener $0{,}0056789 \times 10^5$. Se puede ahora sumar x con el valor desnormalizado de y.

$$\begin{array}{r} x = 0{,}1\,2\,3\,4\,5\,0\,0 \times 10^5 \\ + \; y = \underline{0{,}0\,0\,5\,6\,7\,8\,9 \times 10^5} \\ 0{,}1\,2\,9\,1\,2\,8\,9 \times 10^5 \end{array}$$

Este resultado está ya en forma normalizada y no necesita ninguna renormalización posterior. Se observa que la respuesta se expresa con una precisión de 7 cifras significativas mientras que x e y se tienen sólo con 5 cifras significativas. Si el resultado se va a almacenar en un computador, su mantisa se tendría que reducir a 5 cifras significativas después de la coma (ya que se está trabajando con mantisas con 5 dígitos).

Cuando se realizan operaciones aritméticas en coma flotante de forma manual, con frecuencia se recurre a lo que se puede denominar *precisión flotante*. Si se necesita una mayor precisión simplemente se utilizan más dígitos en los cálculos. Los computadores emplean una representación fija para los números en coma flotante de manera que la precisión no se puede aumentar como consecuencia de un cálculo. Para poner esta idea de manifiesto considérese el siguiente ejemplo de suma de dos números binarios en coma flotante.

$$\begin{array}{l} x = 0{,}1\,1\,0\,0\,1 \times 2^4 \\ y = 0{,}1\,0\,0\,0\,1 \times 2^3 \end{array}$$

El exponente de y se debe incrementar en 1 y su mantisa dividirse por 2 (deplazándola un lugar hacia la derecha) para conseguir que ambos exponentes sean iguales a 4.

$$\begin{array}{l} x = 0{,}1\,1\,0\,0\,1 \quad \times 2^4 \\ y = \underline{0{,}0\,1\,0\,0\,0\,1 \times 2^4} \\ 1{,}0\,0\,0\,0\,1\,1 \times 2^4 \end{array}$$

En este caso se ha producido un desbordamiento de la mantisa y se debe proceder a un proceso de renormalización dividiendo la mantisa por 2 e incrementando en 1 el exponente.

$$x + y = 1{,}0\,0\,0\,0\,1\,1 \times 2^4 \quad \rightarrow \quad 0{,}1\,0\,0\,0\,0\,1\,1 \times 2^5$$

Se han ganado también dos cifras significativas lo que fuerza a que se tome algún tipo de acción. Por ejemplo se puede simplemente truncar el resultado para obtener:

$$x + y = 0{,}1\,0\,0\,0\,0 \times 2^5$$

En la Figura 4.63 se da un diagrama de flujo detallado del algoritmo de suma (resta) de números en coma flotante. De este diagrama de flujo conviene hacer las siguientes observaciones:

1) Como en muchas implementaciones el exponente comparte una palabra con la mantisa, es necesario separarlas antes de comenzar el proceso de la suma. Como ya se ha dicho anteriormente esta operación se denomina desempaquetamiento.

2) Si los dos exponentes difieren en más de $p + 1$, donde p es el número de cifras significativas en la mantisa, entonces el número menor es demasiado pequeño para afectar al mayor y por lo tanto el resultado es efectivamente igual al número mayor y no hay que realizar ningún tipo de acción. Por ejemplo, si la mantisa sólo tiene 4 dígitos, no tiene sentido sumar $0{,}1234 \times 10^{20}$ con $0{,}4567 \times 10^2$, porque el segundo sumando no produce ningún efecto.

3) Durante la renormalización se comprueba el exponente para ver si es menor que un valor mínimo o mayor que un valor máximo. Esto corresponde a comprobar el *subdesbordamiento* y el *desbordamiento del exponente* respectivamente. Cada uno de estos casos representa condiciones en las cuales el número está fuera del rango de números que puede manejar el computador. El subdesbordamiento generalmente conduciría a que el número se haga igual a cero, mientras que el desbordamiento resultaría en una condición de error que debe enviarse para su información al sistema operativo.

4) Los símbolos >> y << especifican un desplazamiento del dato que está a su izquierda hacia la derecha e izquierda respectivamente. A la derecha del símbolo se indica el número de bits de desplazamiento.

4.8.9 Redondeo y truncamiento

En el apartado anterior se acaba de ver que algunas de las operaciones en aritmética en coma flotante conducen a un incremento en el número de bits de la mantisa y que se debe utilizar alguna técnica para mantener constante el número de bits de la mantisa. La técnica más simple se llama *truncamiento* y consiste nada más que en eliminar los bits que no se necesitan. Por ejemplo, 0,1101101 cuando se trunca a cuatro dígitos significativos se convierte en 0,1101. Una técnica mucho mejor es el *redondeo*. Si el valor de los dígitos que se pierden es mayor que la mitad del bit menos significativo de los bits retenidos, entonces se añade un 1 al bit menos significativo de los que quedan. Por ejemplo, considérese el redondeo a 4 bits significativos de los números siguientes:

$$0,1\,1\,0\,1\,1\,0\,1 \rightarrow 0,1\,1\,0\,1 + 1 = 0,1\,1\,1\,0$$
$$0,1\,1\,0\,1\,0\,0\,1 \rightarrow 0,1\,1\,0\,1$$

El redondeo se prefiere siempre al truncamiento de una parte debido a que es más preciso y de otra porque da lugar a un error no polarizado. El truncamiento siempre disminuye el resultado lo que lleva a la

aparición de un error sistemático, mientras que el redondeo algunas veces lo reduce y otras lo aumenta. La desventaja del redondeo es que requiere que se efectúe una operación aritmética adicional sobre el resultado.

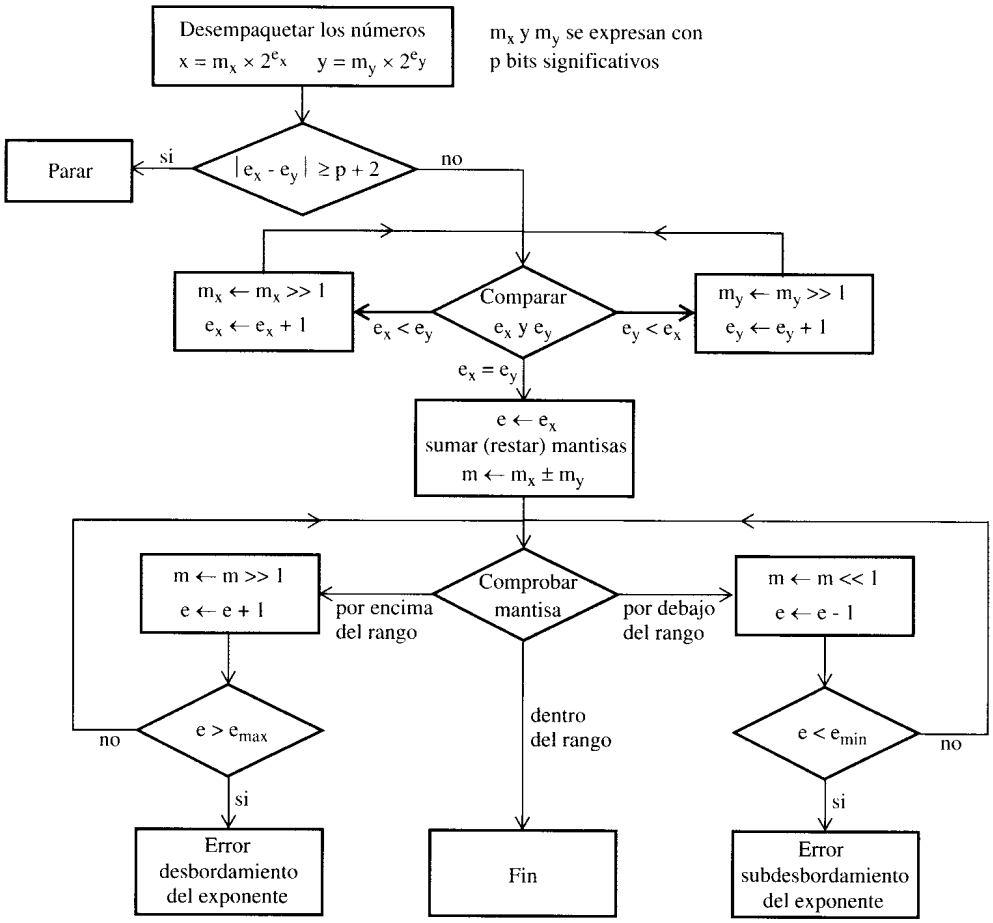


Figura 4.63: Diagrama de flujo del algoritmo de suma (resta) en coma flotante

4.8.10 Algoritmo de multiplicación y división en coma flotante

Se consideran un par de números en coma flotante representados por:

$$x = m_x\, 2^{e_x} \qquad y = m_y\, 2^{e_y}$$

la operación de multiplicación en coma flotante se puede definir como:

$$x \times y = (m_x \times m_y)\, 2^{e_x + e_y}$$

En el caso de la división, la multiplicación de las mantisas se sustituye por su división y la suma de los exponentes por su resta. Por ejemplo, cuando $x = 1{,}000 \times 2^{-2}$ e $y = -1{,}010 \times 2^{-1}$, el producto $x \times y$ se puede calcular como sigue:

1) *Sumar los exponentes*: -2 + (-1) = -3

2) *Multiplicar las mantisas*

```
                1 , 0  0  0
              - 1 , 0  1  0
              -------------
                0  0  0  0
             1  0  0  0
          0  0  0  0
       1  0  0  0
    ---------------
    - 1 , 0  1  0  0  0  0
```

Así el producto es $-1{,}0100 \times 2^{-3}$. En general el algoritmo para la multiplicación (división) es más simple que el de la suma (resta) ya que no precisa ni de la selección del operando con menor exponente, ni del alineamiento de las mantisas en coma flotante. Se distinguen cuatro pasos principales:

1) Realizar la suma (resta) de los exponentes de los operandos.

2) Realizar la multiplicación (división) de las mantisas y determinar el signo del resultado.

3) Normalizar y redondear el resultado si es necesario.

4) Comprobar condiciones de rebose.

La Figura 4.64 muestra la organización típica de un multiplicador (divisor) en coma flotante.

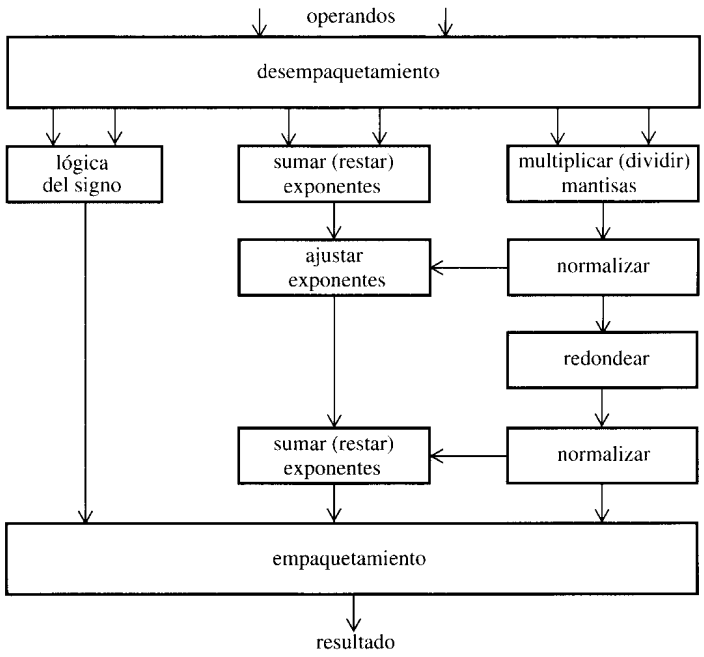


Figura 4.64: Organización de un multiplicador (divisor) en coma flotante

La primera etapa consiste en desempaquetar los operandos: se separan el bit de signo y los campos del exponente y de la mantisa, se reinsertan los bits escondidos de los operandos (en el caso de que el formato utilizado los posea) y se comprueba si hay condiciones de excepción en los operandos (por ejemplo mantisa igual a cero, necesidad de un desplazamiento de una de las mantisas para su alineamiento mayor que el número p de cifras significativas que se emplean en su representación, etc). La comprobación en esta etapa de que una mantisa es igual a cero puede ser útil porque si la operación es la de multiplicación queda ya determinado que el resultado de la operación aritmética es cero, también es necesaria para evitar una inserción impropia de un bit escondido. En la segunda etapa se realiza el cálculo del signo del resultado, la suma (resta) de los exponentes y la multiplicación (división) de las mantisas. A continuación el resultado se normaliza, redondea y renormaliza (si ocurre una condición de rebose durante el redondeo). La última etapa combina los diferentes campos, elimina el bit escondido y comprueba si se han producido condiciones de excepción tales como resultado igual a cero, desbordamiento y subdesbordamiento En la Figura 4.65 se muestra un diagrama de flujo del algoritmo de multiplicación (división) en coma flotante.

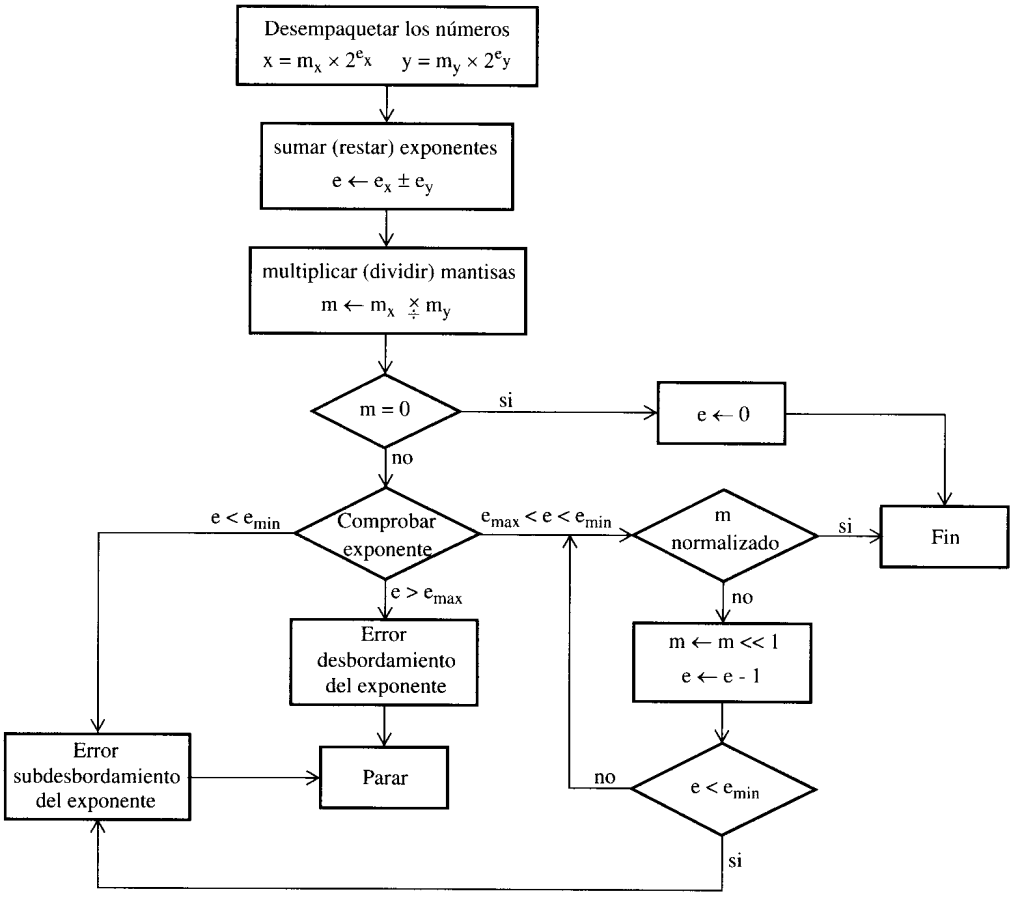


Figura 4.65: Diagrama de flujo del algoritmo de multiplicación (división) en coma flotante

4.8.11 Estructura básica de una unidad aritmética en coma flotante

Una unidad aritmética en coma flotante se puede realizar conectando dos ALU's en coma fija tal como se muestra en la Figura 4.66. La unidad de tratamiento de mantisa se necesita para realizar las cuatro operaciones básicas sobre las mantisas, y se puede utilizar una ALU en coma fija de propósito general del tipo descrito en la Figura 4.55. Para la unidad de tratamiento de exponente es suficiente con un circuito más simple que sea capaz de sumar, restar y comparar exponentes. La comparación de los exponentes, como se ha comentado ya, se puede hacer con un comparador o simplemente restando los exponentes (ver sección 4-10 para una explicación más amplia de la operación de comparación).

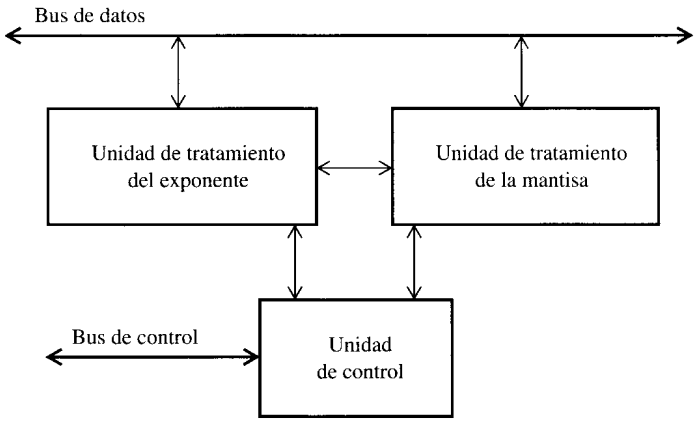


Figura 4.66: Estructura básica de una ALU en coma flotante

En la Figura 4.67 se presenta la estructura general de una ALU en coma flotante que utiliza como comparación la resta de exponentes.

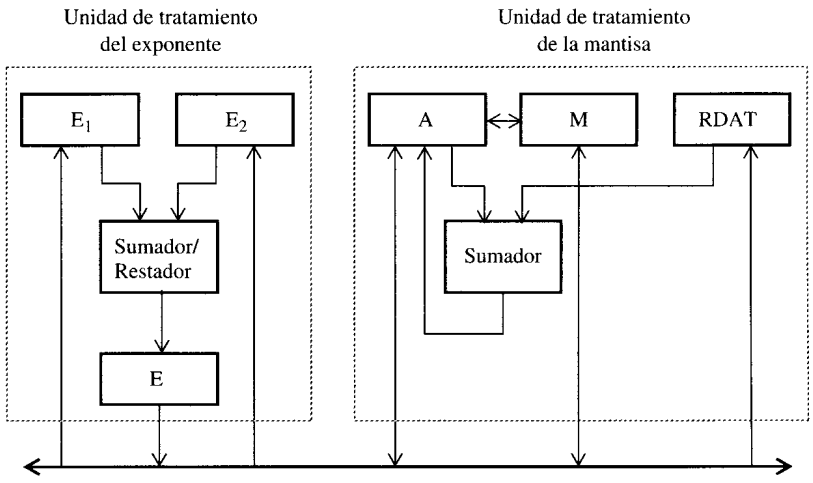


Figura 4.67: ALU en coma flotante

Los exponentes de los operandos de entrada se almacenan en los registros E_1 y E_2, que están conectados a un sumador paralelo que permite calcular $E_1 \pm E_2$. La comparación de los exponentes, que se necesita para la suma y la resta en coma flotante, se efectúa calculando $E_1 - E_2$ y almacenando el resultado en el registro E. El mayor de los exponentes queda determinado por el signo de E. El registro E puede además controlar el desplazamiento que se necesita de una de las mantisas para conseguir sus alineamientos (antes de que pueda tener lugar su suma o su resta). El contenido del registro E se va decrementado de forma secuencial hasta 0. Después de cada decremento se desplaza un dígito la mantisa apropiada. Una vez conseguido el alineamiento de las mantisas, se procesan de la forma normal. Finalmente se calcula el exponente del resultado y se almacena en el registro E.

Sección 4-9

Operaciones de desplazamiento

Al estudiar las operaciones aritméticas se ha visto cómo en algunas de ellas se necesitan efectuar desplazamientos en los registros que almacenan los operandos. Hay otras situaciones en las que conviene realizar sobre los datos ciertas transformaciones que implican diferentes tipos de desplazamientos del contenido de los registros que los almacenan. Por ejemplo, en la extracción de campos en instrucciones o datos, o en la identificación del estado de bits individuales en una palabra.

Por este motivo, la ALU de un computador dispone de registros de desplazamientos capaces de realizar una gran variedad de operaciones de este tipo. De forma genérica, se denomina *registro de desplazamiento* a todo registro capaz de efectuar transferencias series entre sus celdas adyacentes.

4.9.1 Clasificación de las operaciones de desplazamiento

Un registro de desplazamiento típico tiene las entradas y salidas que se muestran en la Figura 4.68. Las entradas de control especifican las diferentes operaciones de desplazamiento que se pueden realizar.

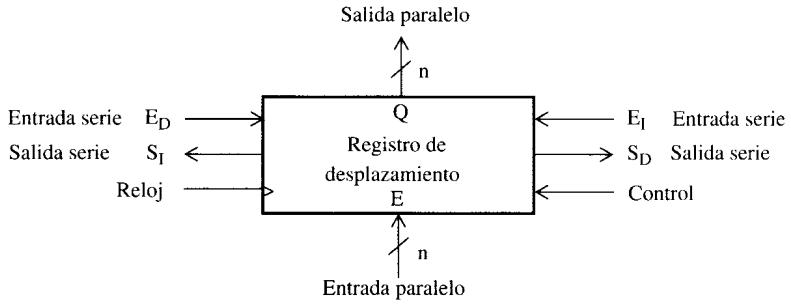


Figura 4.68: Diagrama de bloques de un registro de desplazamiento

Los desplazamientos se pueden clasificar de diferentes formas atendiendo al tratamiento que en cada operación específica se da a la información.

1) *Tratamiento del bit de signo: Aritméticos (A) y Lógicos (L)*

   En los desplazamientos aritméticos no se altera el bit de signo y lo copia en el caso de desplazarse a la derecha. En los desplazamientos lógicos el bit de signo es un bit más en la operación.

2) *Sentido del desplazamiento: Derecha (D) e Izquierda (I)*

3) *Tratamiento de los bits que rebosan: Abierto (A) y Cerrado (C)*

   En los desplazamientos abiertos se pierden los bits que rebosan, mientras que en los desplazamientos cerrados los bits que rebosan por un extremo se introducen por el otro.

4) *Longitud de los registros: Simples (S) y Dobles (D)*

   Cuando la longitud de los registros es simple se utiliza un sólo registro en la operación de desplazamiento y si es doble se emplean dos.

Si se combinan las 4 clasificaciones anteriores resultan 16 variantes de desplazamiento, que se pueden identificar mediante un código de 4 letras tal como se muestra en la Figura 4.69. En la Figura 4.70 se da una representación gráfica de las 16 operaciones de desplazamiento definidas.

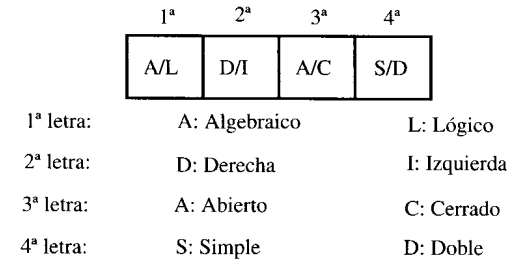


Figura 4.69: Codificación de las operaciones de desplazamiento

4.9.2 Ejemplo: Diseño de un registro de desplazamiento de 4 bits

Se trata de diseñar un registro de desplazamiento de 4 bits del tipo mostrado en la Figura 4.68, capaz de realizar los 2 desplazamientos simples siguientes:

LIAS (Lógico-Izquierda-Abierto-Simple)

LDAS (Lógico-Derecha-Abierto-Simple)

Además de seleccionar las 2 clases de registros de desplazamiento, el circuito debe permitir la carga en paralelo de la entrada y mantener la información almacenada sin modificar (no operación). Por lo tanto, el registro dispone de 4 modos diferentes de funcionamiento y las entradas de control se pueden codificar en 2 líneas (c_2, c_1) de acuerdo con la Tabla 4.19.

Operación	c_2	c_1
NOP	0	0
LIAS	0	1
LDAS	1	0
CARGA	1	1

Tabla 4.19: Codificación de las entradas de control

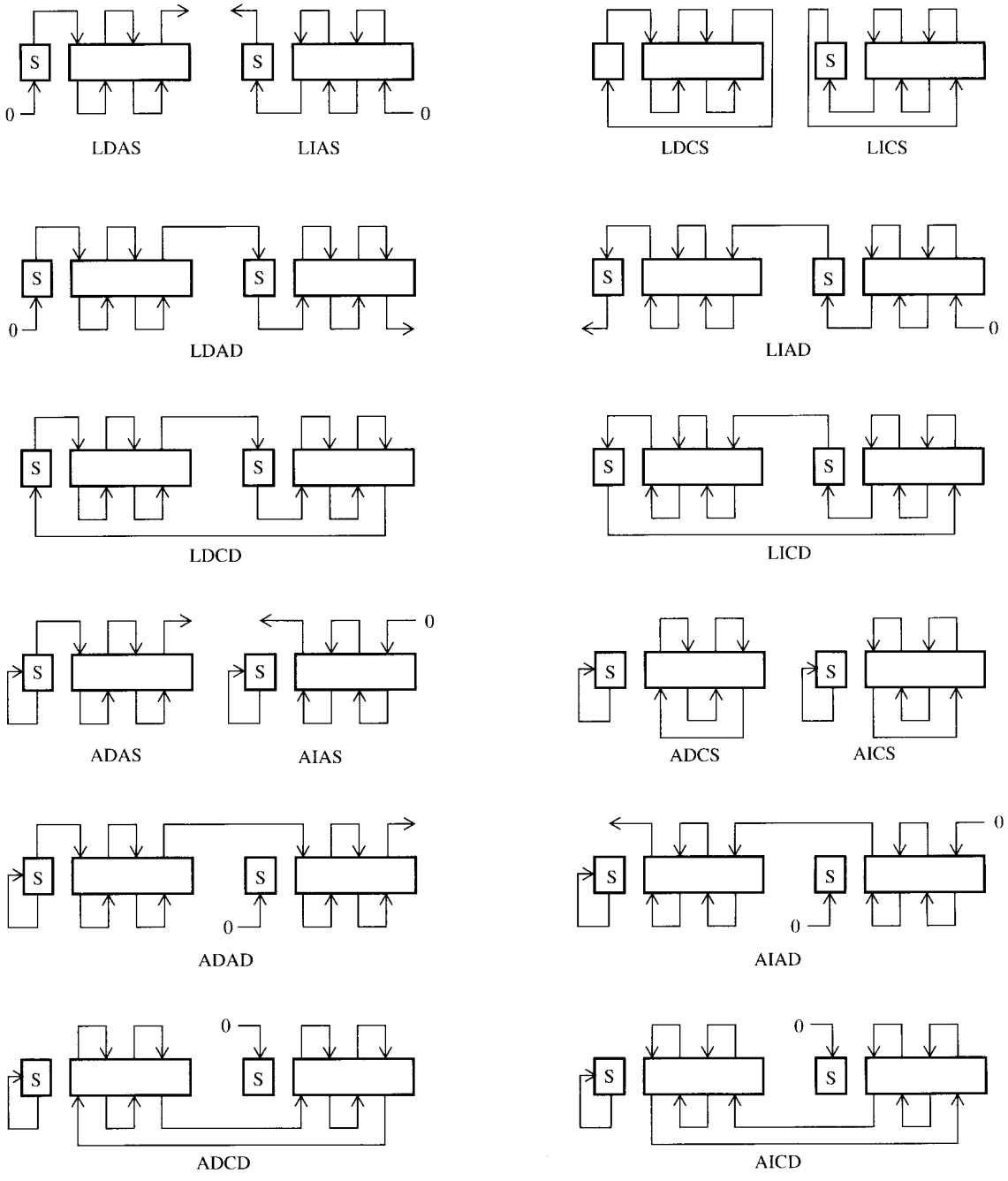


Figura 4.70: Representación gráfica de las operaciones de desplazamiento

La descripción de la salida del circuito es la siguiente:

$$\mathbf{Q}(t+1) = \begin{cases} \mathbf{Q}(t) & \text{si Control = NOP (no operación)} \\ \mathbf{E}(t) & \text{si Control = CARGA} \\ (Q_2, Q_1, Q_0, E_I) & \text{si Control = LIAS} \\ (E_D, Q_3, Q_2, Q_1) & \text{si Control = LDAS} \end{cases}$$

donde $\boldsymbol{Q}(t)$ representa el vector de salidas (Q_3, Q_2, Q_1, Q_0) del registro en el instante t, análogamente $\boldsymbol{Q}(t+1)$ pero en el siguiente período de reloj.

$\boldsymbol{E}(t)$ es el vector de entradas que se desea cargar en el registro desde el exterior y E_D y E_I son entradas series para los desplazamientos a derecha e izquierda respectivamente. En la Figura 4.71 se muestra una de las posibles realizaciones del circuito que utiliza 4 multiplexores para seleccionar, de acuerdo con el valor de las señales de control, cuales son las entradas que se aplican a los elementos de memoria. Así por ejemplo, si $c_2 c_1 = 1\ 0$, las salidas de los multiplexores corresponden a lo que hay aplicado en las entradas número 2, y se ve que en este caso la función que realiza el circuito es un desplazamiento a derecha. ♦

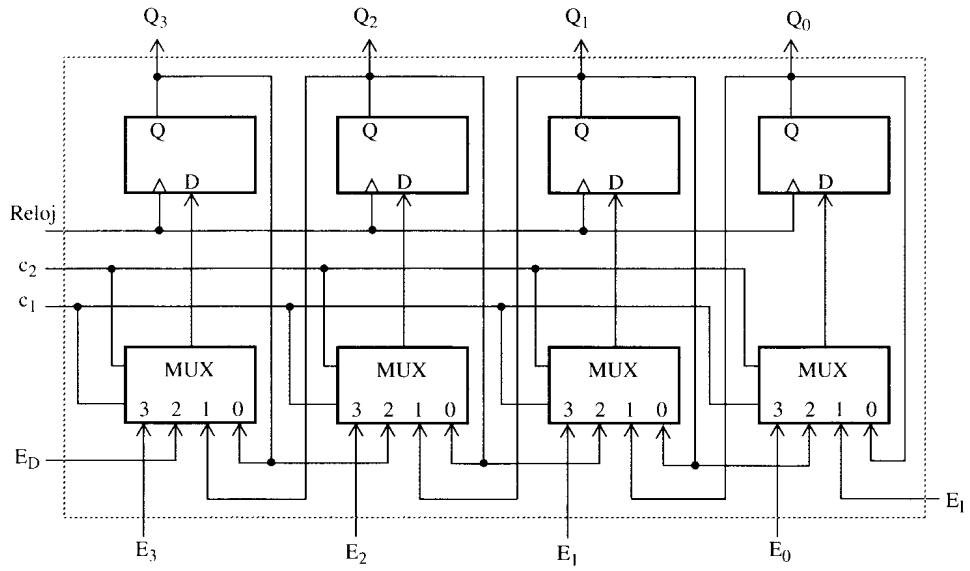


Figura 4.71: Diseño lógico del registro de desplazamiento del ejemplo 4.9.2

4.9.3 Estructura de los registros de desplazamiento

En relación con la forma de procesar la información los registros de desplazamiento se pueden clasificar en 4 categorías:

a) *Entrada paralelo/salida paralelo*

Tanto las entradas como las salidas de todos los bits se producen simultáneamente. El tiempo de ejecución de la operación es independiente de la longitud del registro. En el ejemplo anterior se diseñó un registro de este tipo.

b) *Entrada serie/salida serie*

Dispone de una única entrada E y una única salida Q, tal como se muestra en la Figura 4.72. Por ejemplo, si la operación consiste en un desplazamiento a derecha, en cada período de reloj se obtiene: $Q(t) = E(t - n)$, donde n es la longitud del registro. En este tipo de registro la salida se puede utilizar después de un retardo de n períodos de reloj. A pesar de su lentitud presenta la ventaja de tener un número muy reducido de conexiones externas.

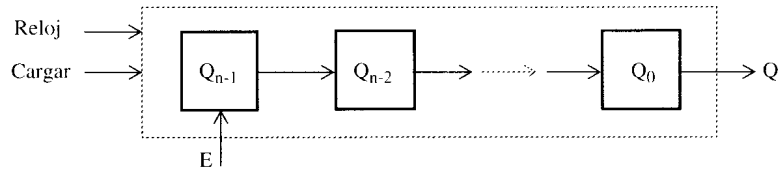


Figura 4.72: Registro de desplazamiento con entrada serie/salida serie

c) *Entrada paralelo/salida serie*

Contiene n entradas E_{n-1}, E_{n-2}, ..., E_0 y una salida Q (ver Figura 4.73). Si la entrada se carga en el registro en $t = 0$ y el registro se desplaza a derecha en cada período de reloj, la salida en t será:

$$Q(t) = E_{t-1}(0) \qquad t > 0$$

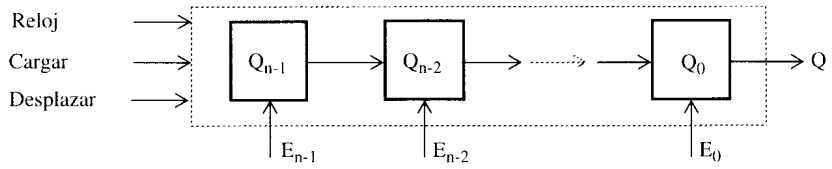


Figura 4.73: Registro de desplazamiento con entrada paralelo/salida serie

d) *Entrada serie/salida paralelo*

Tiene una entrada E y n salidas Q_{n-1}, Q_{n-2},..., Q_0 (ver Figura 4.74). Si se produce un desplazamiento a derecha en cada período de reloj, las salidas son:

$$Q_i(t) = E(t - n + i) \qquad i = 0, 1, ..., n - 1$$

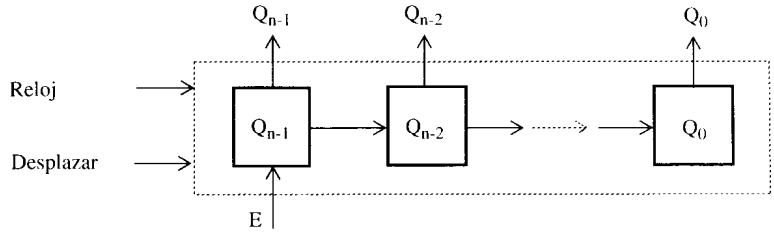


Figura 4.74: Registro de desplazamiento con entrada serie/salida paralelo

Los dos últimos tipos de registros de desplazamientos se suelen utilizar para la interconexión de subsistemas digitales, tal como se muestra en la Figura 4.75. Para reducir los costes de la conexión, los datos que tienen una longitud de n bits se envian desde el sistema S_1 de forma serie utilizando un registro de desplazamiento de "n entradas en paralelo/salida serie". Los n bits transmitidos secuencialmente por S_1 los recibe S_2 en un registro de desplazamiento de "entrada serie/n salidas paralelo".

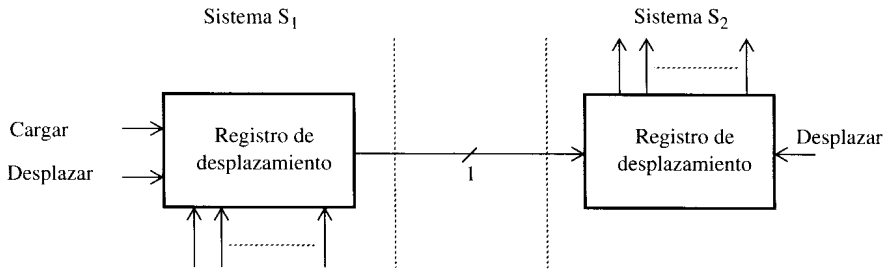


Figura 4.75: Conexión serie utilizando registros de desplazamiento

Sección 4-10

Operaciones de comparación

Una operación lógica que incorporan la mayoría de las ALU's es la comparación de dos operandos. Un comparador es un circuito lógico que tiene dos entradas (x e y), de n bits cada una, que representan las magnitudes de dos números enteros y tres salidas M (mayor), I (igual) y m (menor), que indican la relación que existe entre dichas magnitudes (ver Figura 4.76):

$$\begin{array}{llll} M=1 & I=0 & m=0 & \text{si } x>y \\ M=0 & I=1 & m=0 & \text{si } x=y \\ M=0 & I=0 & m=1 & \text{si } x<y \end{array}$$

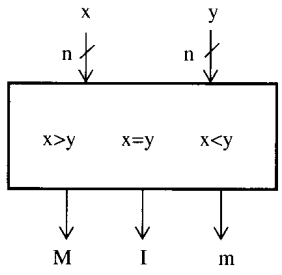


Figura 4.76: Diagrama de bloques de un comparador

El diseño de un circuito comparador se puede realizar de tres formas distintas:

1) Utilizando un circuito combinacional
2) Utilizando un circuito secuencial
3) Utilizando un sumador

4.10.1 Utilizando un circuito combinacional

En el caso más general hay que sintetizar tres funciones lógicas (Mayor (M), Menor (m) e Igual (I)) de $2n$ variables. La complejidad y el coste del circuito aumenta de forma considerable con n.

Una alternativa consiste en sintetizar circuitos comparadores con n pequeño e interconectarlos entre sí. Cuando los números x e y son de un bit cada uno, las señales M, I y m se obtienen de la siguiente tabla de verdad (ver Tabla 4.20):

x	y	M	I	m
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

Tabla 4.20: Tabla de verdad de un comparador de 1 bit

de donde se deducen las siguientes expresiones booleanas para *M*, *I* y *m*:

$$I = x\ \overline{y}$$

$$I = \overline{x}\ \overline{y} + x\ y = \overline{x\ \overline{y} + \overline{x}\ y}$$

$$m = \overline{x}\ y$$

y el correspondiente circuito lógico (ver Figura 4.77).

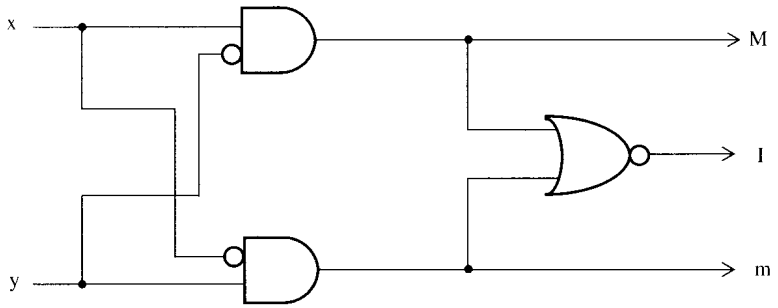


Figura 4.77: Circuito lógico de un comparador de 1 bit

En el caso de números de *n* bits, en lugar de efectuar una síntesis directa mediante un circuito lógico en dos niveles, se puede utilizar una generalización del circuito anterior. Sean *x* e *y* definidos por:

$$x = (x_{n-1},\ x_{n-2},\ ...,\ x_1,\ x_0) \qquad y = (y_{n-1},\ y_{n-2},\ ...,\ y_1,\ y_0)$$

la comparación de cada par de bits x_i e y_i (i = 0, 1, ... n-1) se realizan con comparadores de 1 bit, lo que produce las señales

$$M_i = 1 \qquad \text{si} \qquad x_i > y_i$$

$$I_i = 1 \qquad \text{si} \qquad x_i = y_i$$

$$m_i = 1 \qquad \text{si} \qquad x_i < y_i$$

A partir de M_i, I_i y m_i se generan las salidas *M*, *I* y *m* del comparador de *n* bits mediante las siguientes funciones booleanas:

$$M = M_{n-1} + I_{n-1}\, M_{n-2} + I_{n-1}\, I_{n-2}\, M_{n-3} + ...+ I_{n-1}\, I_{n-2}...I_1\, M_0$$

$$I = I_{n-1}\, I_{n-2}.....I_1\, I_0$$

$$m = m_{n-1} + I_{n-1}\, m_{n-2} + I_{n-1}\, I_{n-2}\, m_{n-3} + ...+ I_{n-1}\, I_{n-2}...I_1\, M_0$$

la explicación de estas expresiones es inmediata, así

$$x > y \quad \text{si} \quad (x_{n-1} > y_{n-1}) \quad \text{ó} \quad ((x_{n-1} = y_{n-1})\ y\ (x_{n-2} > y_{n-2})) \quad \text{ó} \ldots.$$

y análogamente para las otras salidas.

Los circuitos comparadores tienen además 3 entradas de expansión M_{-1}, I_{-1} y m_{-1} con la finalidad de poder realizar las ecuaciones anteriores mediante la asociación de circuitos comparadores.

4.10.2 Utilizando un circuito secuencial

El comparador recibe los bits x_i e y_i de forma serie, comenzando por los más significativos ($i = n\text{-}1, n\text{-}2, ..., 0$). El circuito parte de un estado inicial en el que considera que los dos números son iguales y continúa en este estado mientras $x_i = y_i$.

Cuando detecta que $x_i \neq y_i$ puede decidir si $x > y$ ($x_i = 1$, $y_i = 0$) ó $x < y$ ($x_i = 0$, $y_i = 1$). A partir de ese instante el circuito no cambia de estado. En la Figura 4.78 se representa el diagrama de estados del comparador secuencial. El coste del comparador secuencial es independiente de la longitud n de los números, sin embargo el tiempo que se emplea depende linealmente de n.

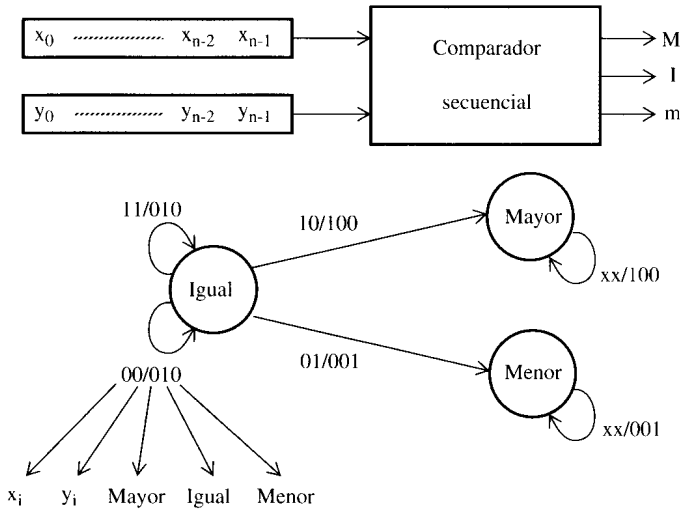


Figura 4.78: Comparador secuencial

4.10.3 Utilizando un sumador

Como la ALU de un computador siempre dispone de un circuito para realizar las operaciones de suma/resta, la forma más usual de realizar la comparación de dos operandos es restando uno de otro y comprobando el signo del resultado.

Los operandos x e y se comparan en una ALU de n bits que sólo dispone de un sumador, por lo que

$$r = x - y = x + C2(x) = x + (2^n - y) = x - y + 2^n \tag{4.22}$$

La CPU también posee una serie de elementos de memoria, en su registro de estado, capaces de almacenar las condiciones que se producen en la ejecución de las operaciones aritméticas (ver Tabla 4.21).

Arrastre (C)	Resultado $\geq 2^n \Rightarrow C = 1$ Resultado $< 2^n \Rightarrow C = 0$
Rebose (V)	$x_{n-1}\, y_{n-1}\, \overline{r}_{n-1} + \overline{x}_{n-1}\, \overline{y}_{n-1}\, r_{n-1} = 1 \Rightarrow V = 1$ $x_{n-1}\, y_{n-1}\, \overline{r}_{n-1} + \overline{x}_{n-1}\, \overline{y}_{n-1}\, r_{n-1} = 0 \Rightarrow V = 0$
Signo resultado (N)	Resultado $< 0 \Rightarrow N = 1$ Resultado $\geq 0 \Rightarrow N = 0$
Resultado cero (Z)	Resultado $= 0 \Rightarrow Z = 1$ Resultado $\neq 0 \Rightarrow Z = 0$

Tabla 4.21: Registros de condición de las operaciones aritméticas

Todos estos elementos de memoria suelen estar agrupados en un sólo registro denominado *registro de estado* del procesador. La condición $C = 1$ implica que al realizar la operación aritmética se produce un arrastre de salida en el bit más significativo del resultado. Las condiciones de rebose y signo del resultado se emplean cuando se representan números con signo.

$V = 1$ significa que se ha excedido la capacidad de representación numérica de los registros. Se produce siempre que los signos de los operandos son iguales y no coinciden con el signo de la suma, (la suma de dos números positivos (negativos) no puede ser un número negativo (positivo)). El procedimiento difiere según sean números sin signo o con signo.

a) *Comparación de números positivos sin signo.* Al efectuar la operación (4.22) se tiene:

si $x \geq y \Rightarrow r \geq 2^n \Rightarrow C = 1$, si $x < y \Rightarrow r < 2^n \Rightarrow C = 0$

De acuerdo con esto y teniendo en cuenta los resultados de la Tabla 4-17 se pueden dar las condiciones que determinan la relación existente entre x e y. En la Tabla 4.22 se muestran las condiciones de comparación en el caso de números positivos sin signo.

Operación: x - y	Condición
$x \geq y$	$C = 1$
$x < y$	$C = 0$
$x > y$	$C = 1$ y $Z = 0 \Rightarrow (\overline{C} + Z = 0)$
$x \leq y$	$C = 0$ ó $Z = 1 \Rightarrow (\overline{C} + Z = 1)$
$x = y$	$Z = 1$
$x \neq y$	$Z = 0$

Tabla 4.22: Comparación de números positivos sin signo

b) *Comparación de números con signo representados en complemento a 2.* En general, al efectuar la operación $r = x - y$ el signo del resultado indicará cual es el mayor de los operandos, es decir:

$$\text{si } r \geq 0 \Rightarrow x \geq y \Rightarrow N = 0 \quad \text{(resultado positivo o cero)}$$
$$\text{si } r < 0 \Rightarrow x < y \Rightarrow N = 1 \quad \text{(resultado negativo)}$$

Sin embargo, el signo del resultado puede no ser el correcto cuando se produce la condición de rebose. Esta condición se detecta por la ALU poniendo el bit de rebose *V* a 1. Por consiguiente cuando V = 1, el signo del resultado de la operación no es el indicado por *N* sino que es el contrario. En consecuencia, al hacer la operación r = x - y, se puede asegurar que:

$$r \geq 0 \Rightarrow x \geq y \text{ si } (N = 0 \text{ y } V = 0) \text{ ó } (N = 1 \text{ y } V = 1) \Rightarrow N \oplus V = 0$$
$$r < 0 \Rightarrow x < y \text{ si } (N = 1 \text{ y } V = 0) \text{ ó } (N = 0 \text{ y } V = 1) \Rightarrow N \oplus V = 1$$

De acuerdo con esto, en la Tabla 4.23 se dan las condiciones que permiten calcular la relación existente entre *x* e *y* cuando representan números con signo en complemento a 2. Compárense los resultados con el caso de números positivos sin signo dados en la Tabla 4.22.

Operación: x - y	Condición
$x \geq y$	$N \oplus V = 0$
$x < y$	$N \oplus V = 1$
$x > y$	$Z = 0$ y $(N \oplus V) = 0 \Rightarrow (Z + (N \oplus V)) = 0$)
$x \leq y$	$Z = 1$ ó $(N \oplus V) = 1 \Rightarrow (Z + (N \oplus V)) = 1$)
$x = y$	$Z = 1$
$x \neq y$	$Z = 0$

Tabla 4.23: Comparación de números representados en complemento a 2

Sección 4-11

Conclusiones

El objetivo básico de la CPU (unidad aritmético-lógica + unidad de control) es buscar instrucciones en una memoria externa y ejecutarlas. Las funciones que realiza una CPU están definidas por su repertorio de instrucciones, que dan una medida de la potencialidad del procesador. En los procesadores más simples las operaciones aritméticas están limitadas a la suma y a la resta de números en coma fija. Los procesadores más complejos son también capaces de multiplicar y dividir y en muchos casos sobre números representados en coma flotante.

La unidad aritmético-lógica (ALU) es el elemento del computador encargado de realizar operaciones con los datos que recibe siguiendo las órdenes que le envía la unidad de control. Lo que la ALU hace en definitiva es sumar, restar, multiplicar, dividir, comparar, desplazar, etc cuando se le suministra la secuencia correcta de señales de control. Es función de la memoria el proporcionar a la ALU los datos que van a utilizarse.

Aunque en la actualidad la ALU puede realizar muchas operaciones, las de tipo aritmético más básicas (suma, resta, multiplicación y división) junto con algunas operaciones lógicas bastante sencillas (AND, OR, desplazamientos, ...) suelen ser las más utilizadas. Las operaciones lógicas son sumamente simples, en comparación con las de tipo aritmético, y se realizan bit a bit sin necesidad de tener en cuenta ningún tipo de arrastres.

En este tema se ha presentado el diseño de los circuitos aritméticos que se utilizan en la mayoría de las ALU's, así como los algoritmos de las operaciones ariméticas tanto en coma fija como en coma flotante. Teóricamente se podría utilizar un circuito combinacional de dos niveles para realizar cualquiera de las operaciones aritméticas básicas, sin embargo su coste sería prohibitivo incluso con longitudes moderadas de los operandos.

El objetivo de diseño se plantea con frecuencia como un compromiso entre la velocidad del circuito y el coste asociado. Cuando se necesitan circuitos aritméticos de muy alta velocidad es necesario emplear circuitos combinacionales, aunque es preciso considerar el problema de los arrastres que surgen en todas las operaciones aritméticas. Por este motivo se analizan diversas estrategias para acelerar la suma y la multiplicación. En todas ellas la finalidad es acelerar la generación de los arrastres para garantizar el resultado correcto en el menor tiempo posible.

En el caso de la multiplicación y de la división en coma fija se pueden emplear algoritmos secuenciales que son similares a los que se utilizan en los métodos manuales. El producto o cociente de dos números de n bits de longitud se puede formar en n pasos secuenciales, donde en cada paso se efectúa un desplazamiento y posiblemente una suma o resta. La división es más difícil que la multiplicación debido a que hay que determinar los dígitos del cociente. Finalmente se han estudiado las operaciones de desplazamiento y de comparación que son esenciales en cualquier ALU.

Sección 4-12

Problemas

1) Realizar un estudio análogo al del SSB para el semirestador binario (SRB). Consta de dos entradas binarias (x e y) y dos salidas; una de ellas (d) es el resultado de la "diferencia módulo 2" de las dos entradas y la otra (c) es el "arrastre" de la resta.

2) Diseñar un circuito conjunto SSB-SRB. El circuito dispone de una señal de control c, de forma que cuando $c = 0$ sus salidas corresponden a un SSB y cuando $c = 1$ a un SRB.

3) Realizar un estudio análogo al del SBC para el restador binario completo (RBC). Consta de tres entradas binarias (x_i, y_i y c_{i-1}) y dos salidas; una de ellas $d_i = x_i - y_i - c_{i-1}$ y la otra c_i es el "arrastre" de la resta para la etapa siguiente.

4) Generalizar la celda básica del SBC transformándola en un "Sumador-Restador Controlado" (SRC). La entrada adicional de control p, se utiliza para definir las dos operaciones: $p = 0 \Rightarrow$ suma y $p = 1 \Rightarrow$ resta.

5) Completar el sumador-restador de la Figura 4.13 con un circuito lógico que corrija el signo de la suma cuando éste se produce erróneamente.

6) Diseñar un sumador-restador para números con signo representados en complemento a 1. Incluir el circuito de detección de rebose.

7) Diseñar un sumador-restador para números con signo representados en magnitud-signo. Incluir el circuito de detección de rebose.

8) Demostrar la equivalencia de las expresiones (4.3) y (4.4).

9) Cada una de las siguientes parejas de números binarios se suman en un sumador binario paralelo con propagación de arrastres. En cada caso determinar: 1) El número de secuencias de arrastres que comienzan simultáneamente. 2) La longitud de la secuencia de arrastre más grande.

Pareja 1: 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 1
0 1 0 0 1 1 1 1 0 1 0 0 1 0 1 1

Pareja 2: 0 1 0 0 1 0 0 1 0 1 0 1 1 0 1 1
0 1 0 1 0 1 1 0 1 1 0 1 0 0 1 0

10) Efectuar la suma de las dos parejas de números del ejercicio anterior utilizando un sumador de suma condicional.

11) Calcular el retardo en un sumador de n bits construido con k módulos SBAA de m bits ($n = k \times m$ bits).

12) Comparar, para el peor caso, la aceleración que se obtiene con un sumador con aceleración de arrastre a 2 niveles para 4 grupos de 4 bits (longitud de palabra = 16 bits) frente al sumador binario paralelo correspondiente. Dibujar sus diagramas de bloques.

13) Diseñar un sumador de 15 números de 1 bit de longitud utilizando únicamente módulos SBC's.

14) Demostrar, de forma general, que en un sumador BCD se debe sumar 6 cuando hay que corregir el resultado producido por el primer sumador binario.

15) Deducir la ecuación (4.10) correspondiente a las funciones booleanas del complementador a 9 en BCD.

16) Diseñar un sumador de dos dígitos decimales representados en código *exceso-3*.

17) Demostrar que el producto de dos números, x e y, con n dígitos en base B no tiene más de $2n$ dígitos.

18) Diseñar un multiplicador binario de 2 × 2 bits utilizando 4 puertas AND y 2 SSB's.

19) Diseñar un multiplicador binario de 4 × 3 bits, utilizando 12 puertas AND y 2 sumadores binarios paralelo de 4 bits cada uno.

20) Diseñar un multiplicador de 4 × 8 bits, utilizando 2 memorias ROM de $2K \times 8$ bits y 2 sumadores binarios paralelo de 4 bits cada uno.

21) Mostrar paso a paso el proceso de multiplicación, utilizando los algoritmos de "lápiz y papel mejorado" y de Booth cuando se multiplican los siguientes números binarios con signo:

1) $(+15)\times(+13)$ 2) $(+15)\times(-13)$

(suponer que los registros que almacenan los números tienen una longitud de 5 bits).

22) Calcular el retardo de un multiplicador de $n \times n$ bits que utiliza la estructura de la Figura 4.43.

23) Demostrar la expresión (4.18), en un multiplicador de $n \times n$ bits que utiliza un árbol de Wallace para reducir la altura de las columnas de la matriz de productos parciales a 2.

24) Demostrar la regla práctica dada para obtener el complemento a 2 de un número.

25) Deducir la tabla de codificación del algoritmo de Booth modificado cuando se examinan 4 dígitos simultáneamente. Proponer en este caso la estructura del multiplicador.

26) Utilizando el método de dividir de "lápiz y papel" efectuar las siguientes divisiones: 41÷3 y 73÷5. Suponer que el dividendo tiene una longitud de 8 bits y el divisor de 4.

27) Efectuar las siguientes divisiones mediante el método de restauración:

Dividendo = 13 = $0\,0\,0\,0\,1\,1\,0\,1_2$ Divisor = 7 = $0\,1\,1\,1_2$; Dividendo = 19 = $0\,0\,0\,1\,0\,0\,1\,1_2$ Divisor = 9 = $1\,0\,0\,1_2$

28) Realizar el mismo ejercicio anterior por el método de no restauración.

29) Diseñar una ALU que efectúe las siguientes combinaciones de operaciones:

a) NAND, NOR, transferencia y complemento.

b) XOR y XNOR.

30) Diseñar una ALU de 4 bits que realice las operaciones aritméticas que se muestran en la Tabla 4.24.

Operaación	Descripción
R = A + B	Suma A y B
R = A - B	Resta B de A
R = $\overline{B}$	C1(B)
R = $\overline{B}$ + 1	C2(B)
R = A + $\overline{B}$ + 1	Suma A y C2(B)
R = A + 1	Incrementa A en 1
R = A - 1	Decrementa A en 1

Tabla 4.24: Operaciones aritméticas de la ALU

31) Diseñar una ALU de 4 bits que realiza las operaciones lógicas que se muestran en la Tabla 4.25.

Operación	Descripción
R = A $\wedge$ B	AND de A y B
R = A $\vee$ B	OR de A y B
R = A $\oplus$ B	XOR de A y B
R = $\overline{A}$	C1(A)

Tabla 4.25: Operaciones lógicas de la ALU

32) Combinar las dos ALU's de los ejercicios anteriores en una única ALU.

33) Utilizando el circuito integrado *SN74x181* (ver Figura 4.56) construir un sumador-restador de 16 bits para números representados en complemento a 2.

34) Realizar las suma de x e y en coma flotante en los siguientes casos:

a) x=00001111 y=00000101; b) x=00111011 y=00000110

Los operandos deben expresarse como fracciones normalizadas y la suma debe darse también normalizada.

35) ¿Bajo que condiciones se produce un rebose del exponente durante una operación de suma en coma flotante?

36) Demostrar que no puede existir rebose de la mantisa en una operación de multiplicación en coma flotante.

37) Demostrar que la división en coma flotante de dos números con mantisas normalizadas da un cociente que ya está normalizado.

38) Sumar 10,125 y 32,1 utilizando aritmética en coma flotante con el formato que se muestra en la Figura 4.79. En cada caso mostrar como se almacenarían los números en el computador.

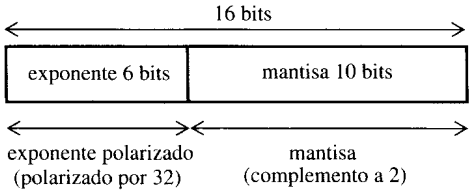


Figura 4.79: Formato de los números en coma flotante

39) Diseñar un registro de desplazamiento de 8 bits del tipo entrada paralelo/salida paralelo, capaz de realizar los dos desplazamientos dobles siguientes:

LIAD (Lógico-Izquierda-Abierto-Doble)

LDAD (Lógico-Derecha-Abierto-Doble)

Utilizar como módulo básico el registro diseñado en el Ejemplo 4.9.2

40) El registro de desplazamiento de 4 bits del Ejemplo 4.9.2 tiene una estructura de entrada paralelo/salida paralelo. Diseñar el mismo registro pero con las siguientes estructuras: 1) entrada serie/salida serie. 2) entrada paralelo/salida serie. 3) entrada serie/salida paralelo.

41) Diseñar un circuito de comparación combinacional de números de 2 bits de longitud.

42) Diseñar el circuito de comparación secuencial correspondiente al diagrama de estados de la Figura 4.78.

43) Diseñar las versiones serie y paralelo de un comparador que pueda comparar números representados en signo-magnitud.

44) Utilizando una ROM de 16 × 4 diseñar un comparador de 2 bits que genere las funciones "mayor que", "menor que" e "igual que".

45) Se supone una longitud de palabra de 8 bits y que los números que se representan son sin signo. Verificar las relaciones entre *x* e *y*, utilizando el método de comparación por suma en los dos casos siguientes:

1) x = 57 e y = 23

2) x = 23 e y = 57

46) Se supone una longitud de palabra de 8 bits y que los números se representan en C2. Verificar las relaciones entre *x* e *y*, utilizando el método de comparación por suma en los dos casos siguientes:

1) x = -57 e y = 23

2) x = 23 e y = -57

Tema 5

Diseño de transferencia entre registros

Un *sistema* se puede definir de manera informal como una colección de objetos, denominados *componentes*, que se conectan de forma coherente con un objetivo bien definido. La función que realiza el sistema queda determinada cuando se conoce:

a) La función individual de cada componente

b) La forma en que los componentes se interconectan entre sí

Desde este punto de vista, se considera que un *sistema digital* tiene como misión transformar un conjunto X de datos de entrada (por ejemplo un programa con sus instrucciones y sus datos) en un conjunto Y de resultados de salida. La transformación se puede expresar formalmente por $Y = F(X)$.

Una forma bastante natural y útil de modelar un sistema es mediante un grafo. Un grafo contiene un conjunto de vértices, V, llamados *nodos* y un conjunto de arcos, A, cuyos elementos son pares de elementos de V. Si el grafo es dirigido los elementos de A son pares ordenados (ver Figura 5.1).

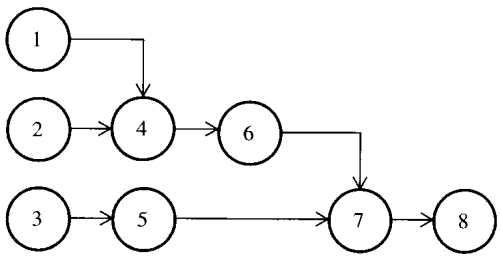


Figura 5.1: Grafo dirigido con 8 nodos y 7 arcos

Un sistema se puede considerar que está constituido por dos clases de elementos:

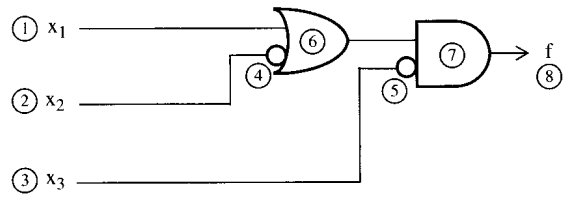
a) Un conjunto de componentes, C, para el procesamiento de la información.

b) Un conjunto de señales, S, que transmiten la información entre las componentes.

Cuando se modela un sistema digital, se efectúa la correspondencia entre C y los nodos y S y los arcos. Al grafo que se obtiene de esta manera se le denomina *diagrama de bloques* del sistema. Un diagrama de bloques de esta naturaleza sirve para modelar, en principio, la *estructura del sistema*, pero no vale para mostrar su *conducta funcional*. Por este motivo, en un diagrama de bloques se suele explicitar el comportamiento de cada nodo por medio de algún criterio o convenio establecido a priori. Así, si se conoce la conducta de cada uno de los nodos se puede conocer el comportamiento funcional global del sistema. En la Figura 5.2 se muestran algunas formas, que se utilizan normalmente, para describir el comportamiento funcional de un sistema digital como son: tablas de verdad, puertas lógicas y funciones de conmutación.

a)

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



b)

$$f = (x_1 + \bar{x}_2) \bar{x}_3$$

c)

Figura 5.2: Representaciones del comportamiento funcional de un sistema digital
a) Tablas de verdad b) Puertas lógicas c) Funciones de conmutación

El diagrama de bloques de la Figura 5.2b representa un grafo análogo al de la Figura 5.1 y muestra un sistema con 3 entradas, 1 salida y 4 componentes interconectados de una forma determinada. La correspondencia entre los nodos de los dos grafos es la siguiente:

Nodo 1 ↔ Entrada x_1
Nodo 2 ↔ Entrada x_2
Nodo 3 ↔ Entrada x_3
Nodo 4 ↔ Puerta NOT
Nodo 5 ↔ Puerta NOT
Nodo 6 ↔ Puerta OR
Nodo 7 ↔ Puerta AND
Nodo 8 ↔ Función f

El grafo de la Figura 5.2b pone de manifiesto, además de la estructura de interconexión de los componentes, su funcionamiento. *Si no se especifica lo que realiza cada bloque, no se puede saber el comportamiento funcional del sistema.*

En este tema se estudia el procedimiento de diseño de un computador o con mayor generalidad de cualquier sistema digital. El método de diseño puede considerarse a diferentes niveles de detalle o complejidad. Se suelen emplear los tres niveles siguientes:

1) *Nivel de puertas lógicas*. Corresponde al estudio de la teoría de la conmutación y tiene interés para los diseñadores de sistemas. En este nivel la unidad de información es el **bit**.

2) *Nivel de registros*. Un programador en lenguaje máquina o lenguaje ensamblador contempla a un sistema digital desde este nivel. No se tratan ya las puertas lógicas de forma individual, sino los

registros de la máquina y sus interconexiones. Es el nivel que se emplea cuando se consideran las características estructurales de un sistema digital. La unidad de información es la **palabra**.

3) *Nivel de procesador.* Los objetos básicos que se consideran en este nivel son elementos tales como CPU, procesadores de E/S, memorias etc. La característica fundamental de estos módulos es su elevada complejidad. La unidad de información es un **bloque de palabras** que puede representar o un programa o una zona de datos.

En este tema se estudia una metodología general de diseño de sistemas digitales cuando se describen a nivel de transferencia de registros. Se exponen los fundamentos de los lenguajes de descripción que se pueden utilizar tanto en los niveles de especificación como de realización. La descripción de un sistema digital se divide en dos partes: la *unidad de procesamiento* y la *unidad de control* (modelo de Glushkov).

La unidad de procesamiento es la parte del sistema digital en la que se almacenan y transforman los datos. Consta de los siguientes elementos:

a) registros de almacenamiento

b) operadores aritmético-lógicos (recursos de cálculo)

c) red de interconexión (caminos de datos)

d) puntos de control

e) señales de condición

La función de la unidad de control es generar una secuencia de señales de control de acuerdo con el algoritmo de transferencia de registros, que especifica la realización de la operación deseada. En muchos casos, una máquina secuencial es un modelo adecuado para la realización de la unidad de control. En este tema se presenta una forma de carácter gráfico, de especificar el modelo de un sistema secuencial: *el diagrama de máquina de estados algorítmica* o *diagrama ASM* (acrónimo de Algorithmic State Machines). El diagrama ASM se caracteriza por el hecho de que describe una secuencia de sucesos así como la relación temporal entre los estados de la unidad de control y las acciones que ocurren en los estados como respuesta a los distintos pulsos de reloj.

Se analiza con detalle la organización de la unidad de control y se describen diferentes realizaciones de la misma. En concreto se analizan las siguientes alternativas para el diseño de la unidad de control:

1) Elementos de memoria tipo *D*

2) Registro de secuencia y un decodificador

3) Un elemento de memoria por estado

4) Un registro de estado y una memoria ROM

5) Un contador y un decodificador

6) Un registro de estado y un array lógico programable (PLA)

Con el fin de reforzar los conceptos que se van introduciendo a lo largo del tema y a modo de ejemplo ilustrativo se desarrolla de forma detallada la realización de un multiplicador de dos números binarios sin signo de n bits, basado en el algoritmo de *lápiz y papel mejorado* propuesto en el tema 4.

Sección 5-1

Niveles de diseño de un sistema digital: diseño jerárquico

La representación que se utiliza para definir un sistema es diferente según sea el nivel de descripción empleado (procesador, registros o puertas lógicas). Otro aspecto previo a destacar es que la separación entre los distintos niveles no es en absoluto nítida y es normal encontrar descripciones donde se combinan elementos de más de un nivel. El nivel que se va a considerar fundamentalmente es el nivel de registros.

Es costumbre denominar los niveles de diseño como *altos o bajos*, dependiendo de la complejidad de los componentes empleados. Esto da origen a una jerarquía de niveles según el tipo de elementos utilizados. En esta estructura jerárquica un componente de un determinado nivel N_j es equivalente a un conjunto de componentes del nivel inmediatamente inferior N_{j-1} (ver Figura 5.3).

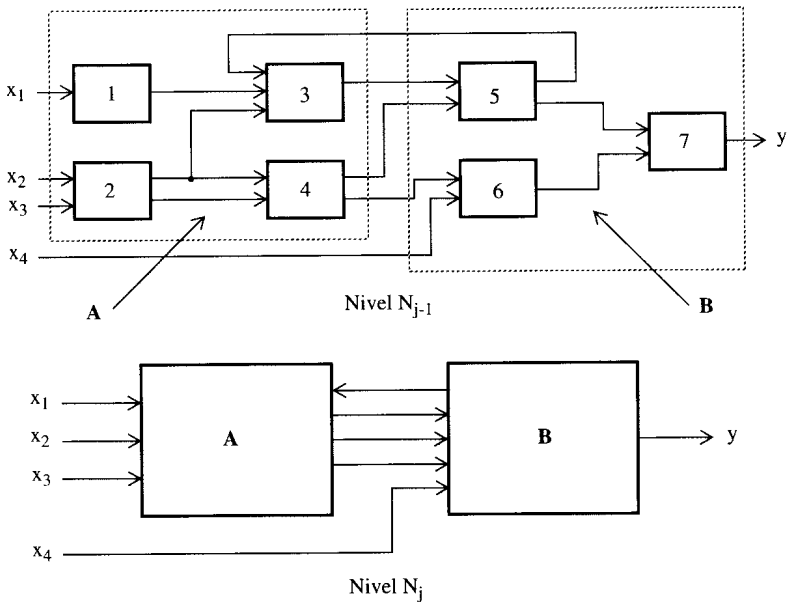


Figura 5.3: Niveles de diseño

Si un sistema complejo se va a diseñar con puertas lógicas, el proceso de diseño constará de los siguientes pasos:

1) Especificar a nivel de procesador la estructura del sistema

2) Especificar a nivel de registros la estructura de cada unidad del nivel de procesador

3) Especificar a nivel de puertas lógicas la estructura de cada componente del nivel de registro

Este punto de vista del diseño se conoce como *diseño descendente* o *diseño estructurado*. Las técnicas de diseño son bastante diferentes según el nivel de diseño de que se trate. Un buen diseño debe promover la utilización de los componentes más adecuados en cada nivel. Desde esta perspectiva, conviene tener en cuenta las siguientes características:

a) *Independencia de los componentes*

Los componentes deben ser tan independientes como sea posible. Esto permite que cada uno de ellos se pueda analizar, diseñar y comprobar de forma aislada.

b) *Modularidad del diseño*

Los límites de separación de los componentes del diseño, a un determinado nivel, deben corresponderse con las fronteras físicas establecidas por la tecnología disponible en cada momento. Esto corresponde a realizar un diseño lo más *modular* posible, de manera que cada componente sea físicamente reemplazable por otro, tal como ocurre con un circuito integrado, una tarjeta de circuito impreso o una unidad completa. Esta forma de diseñar los sistemas facilita su construcción y reparación.

Sección 5.2

Nivel de transferencia entre registros

Como se ha indicado ya, en este nivel la unidad básica es la palabra, es decir, un conjunto ordenado de bits que representa cierta información. Atendiendo al tipo de información se pueden distinguir dos clases de palabras:

a) *Palabras de datos* (por ejemplo los datos de un programa)

b) *Palabras de control* (por ejemplo instrucciones)

Los componentes que se emplean en un cierto nivel de diseño constituyen su principal caracterización. En la Tabla 5.1 se muestran los componentes empleados en el nivel de registro; los elementos que se recogen en ella se pueden encontrar en la gran mayoría de los sistemas digitales.

Componentes combinacionales	Componentes secuenciales	Componentes de comunicación
- Puertas de palabra - Multiplexores - Decodificadores - Codificadores - Dispositivos lógicos programables - Unidades aritméticas	- Biestables - Registros - Contadores	- Buses

Tabla 5.1: Componentes empleados en el diseño a nivel de registro

Una cuestión importante que conviene considerar es la *completitud* de este conjunto de componentes. Un conjunto de componentes se dice que es completo si se puede realizar cualquier función lógica utilizando sólo módulos de ese conjunto. Al nivel de puertas lógicas la teoría de conmutación establece la completitud de determinados conjuntos de puertas (por ejemplo AND, OR y NOT). Sin embargo, al nivel de registro no se tiene un marco teórico que permita definir la noción de completitud y por lo tanto dicha cuestión carece de un sentido preciso. Conviene advertir, que cualquier función de conmutación se puede sintetizar utilizando exclusivamente multiplexores, pero esta propiedad no guarda mucha relación con la aplicación básica de los multiplexores en el diseño a nivel de registro que consiste en actuar como módulos de selección sobre un conjunto de señales. Lo mismo puede decirse de los PLA.

5.2.1 Representación

No existe una representación universal para el diseño a nivel de registro, pero la que se emplea con más frecuencia es la de los *diagramas de bloques*.

En un diagrama de bloques cada componente se visualiza mediante un módulo con sus entradas y salidas, conjuntamente con una descripción de la función que realiza. También se suele efectuar una separación (explícita o implícita) de sus entradas y salidas en:

a) Información de *datos*

b) Información de *control*

Una palabra de datos de n bits se representa mediante una sola línea cruzada por una barra al lado de la cual se escribe el valor n. En la Figura 5.4 se muestra un ejemplo típico de un elemento de un diagrama de bloques a nivel de registro.

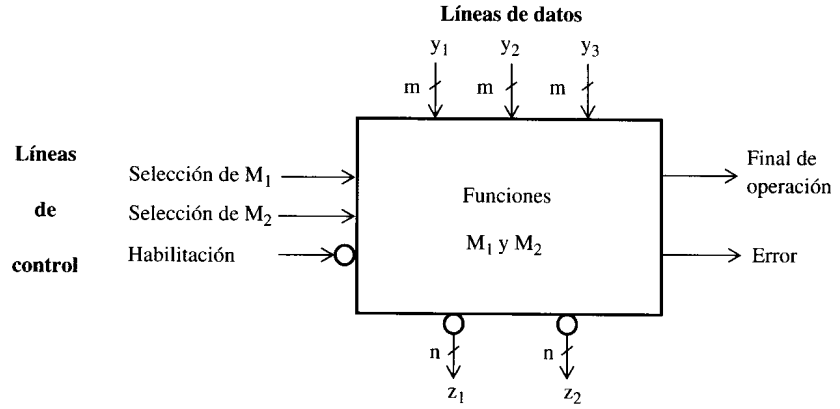


Figura 5.4: Representación de un diagrama de bloques a nivel de registro

No hay símbolos estándar para representar un diagrama de bloques a nivel de registro, de ahí que se suela utilizar un rectángulo tal como se indica en la Figura 5.4. El tipo de función que realiza se incluye en el interior del rectángulo y, en general, las líneas de entradas y_1, y_2, ..., y_j y de salidas z_1, z_2, ..., z_k pueden representar buses (conjuntos de entradas y salidas) de diferentes tamaños (n_1, n_2, ...).

El diagrama de la Figura 5.4 representa un bloque capaz de realizar dos funciones M_1 y M_2 sobre un conjunto de 3 datos de entrada y_1, y_2 e y_3 de m bits cada uno, generando dos datos de salida z_1 y z_2 de n bits cada uno. Las líneas de control aparecen etiquetadas de manera que se identifica cuál es su misión. Se pueden clasificar en dos grandes grupos:

1) *Líneas de selección.* Su misión es seleccionar uno de los posibles modos de funcionamiento que posee el circuito.

2) *Líneas de habilitación.* Permiten, mediante el posicionamiento en un determinado nivel lógico, que se especifique el instante de tiempo en que debe realizarse la operación previamente seleccionada.

Por ejemplo, si se desea conseguir que el circuito realice la función M_2 se deberá hacer lo siguiente:

a) Posicionar la entrada *Selección* M_2 en el valor "1 lógico". Este valor se conoce como estado activo de la línea.

b) Posicionar la entrada *Habilitación* en el valor "0 lógico". El pequeño círculo que aparece en el punto de entrada de la línea de habilitación del circuito quiere decir que el estado activo de esta línea es el "0 lógico".

5.2.2 Expandibilidad de los componentes

En general, un diseño al nivel de transferencia entre registros puede necesitar componentes de diferentes longitudes de palabra. Por este motivo, conviene utilizar módulos de un determinado tipo (por ejemplo registros) con entrada de n bits, para diseñar módulos del mismo tipo con entrada de m bits, siendo $n \neq m$. Se pueden presentar dos casos:

1) $m < n$. El problema se resuelve de forma sencilla, ya que basta con no usar las entradas que sobran del módulo de n bits, conectándolas a unos valores lógicos constantes adecuados.

2) $m > n$. Hay que diseñar el circuito deseado mediante la conexión de algunos módulos de n bits y quizás componentes de otros tipos. Esta posibilidad es lo que se conoce como *expandibilidad de los componentes*. Al nivel de registro los componentes deben diseñarse de tal manera que su ampliación sea relativamente simple.

Sección 5-3

Estructura de un sistema digital

El comportamiento de un sistema digital a nivel de transferencia de registros se puede definir mediante un conjunto de operaciones O_1, O_2, ..., O_n realizadas sobre diferentes datos externos o internos (memorias registros). Cada una de estas operaciones O_i se realiza mediante una secuencia de operaciones elementales de transferencias entre registros descritas por relaciones del siguiente tipo:

$$R[j] \leftarrow f(R[i]) \tag{5.1}$$

donde los datos colocados en la posición i son transformados por una función lógica o aritmética f y donde el resultado de esta transformación se envía a una posición j (i y j pueden tener el mismo valor, ver Figura 5.5).

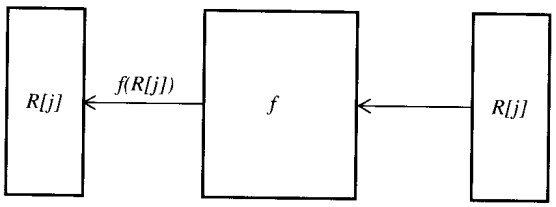


Figura 5.5: Operación elemental de transferencia entre registros

De manera general la funcion f puede actuar sobre varios datos distintos:

$$R[m] \leftarrow f(R[i], R[j], R[k], ...) \tag{5.2}$$

La operación elemental (5.2) permite definir los siguientes conceptos:

- Se llama *microoperación* (μ_{op}) a toda operación del tipo (5.2) realizada por el procesador en un sólo período de la señal de reloj.
- Los datos $R[i]$, $R[j]$, $R[k]$, ... se llaman los *operandos*
- La función f es el *operador* de la μ_{op} que representa una operación realizable por uno de los componentes o *recursos de cálculo* que posee la máquina (por ejemplo un sumador). De esta manera, *la secuencia de transferencias entre registros que realizan O_i constituyen el algoritmo de O_i.*

La operación particular O_i a realizar por un sistema digital puede ser seleccionada por una señal de control externa. Así, por ejemplo, el registro de desplazamiento de la Figura 4.68 dispone de dos señales de control. Cuando el sistema digital es capaz de efectuar diferentes operaciones las señales de selección de función están normalmente agrupadas en palabras llamadas *instrucciones*. En estos casos se habla de

procesador de un conjunto de instrucciones. El problema de diseño a nivel de transferencia de registros se puede plantear en los términos siguientes:

- Dados un *conjunto de algoritmos*, un *repertorio de instrucciones* y un *conjunto de restricciones*, obtener la estructura del circuito que realiza las funciones deseadas empleando sólo componentes de este nivel.

Las ligaduras o restricciones especificadas en un problema de diseño, en la mayoría de los casos, imponen un compromiso entre el coste del circuito y su comportamiento (velocidad, fiabilidad, compatibilidad ...). Así por ejemplo la sentencia:

$$a: \quad R_A \leftarrow R_A \times R_B, \qquad R_C \leftarrow R_C \times R_D;$$

define dos operaciones de multiplicación. Ambas operaciones se pueden ejecutar en paralelo ya que actúan sobre operandos diferentes. Esto requiere que necesariamente el sistema digital disponga de dos unidades de multiplicación. Es posible una segunda solución que sólo utiliza un multiplicador (menor coste) a expensas de realizar de forma secuencial las dos multiplicaciones (menor velocidad):

$$a_0: \quad R_A \leftarrow R_A \times R_B;$$

$$a_1: \quad R_C \leftarrow R_C \times R_D;$$

Éste es un ejemplo típico de compromiso coste-comportamiento. La transformación de algoritmos paralelos en secuenciales (o viceversa) establece un mecanismo claro de llevar a cabo estrategias de este tipo. El programa debe indicar al procesador las microoperaciones que van a ser ejecutadas y el orden o secuencia de su ejecución

5.3.1 Componentes de un sistema digital

Las definiciones precedentes suponen la existencia en el sistema digital de los componentes siguientes:

1) *Un conjunto de posiciones de memoria* $R[i]$. Los datos deben estar localizados en unas posiciones precisas en el interior del sistema digital. Cada posición de un dato está identificada por una dirección única. La especificación de una μ_{op} contiene por lo tanto las direcciones de sus operandos. Estos elementos de memoria son los *registros del procesador.*

2) *Un interfaz con el exterior.* Los datos provenientes del exterior deben utilizar este interfaz para comunicarse con el sistema digital. Los datos externos son funcionalmente equivalentes a los registros internos. Cada elemento está identificado por una dirección única y los operandos $R[i]$, $R[j]$, $R[k]$, ... de la expresión (5.2) se aplican igualmente a los registros internos del sistema digital y a los datos externos. Sin embargo ciertas características físicas, principalmente una velocidad de acceso inferior, hacen que su tratamiento sea a menudo diferente. Esta es la razón de la existencia de una interfaz.

3) *Un conjunto de dispositivos lógicos que realizan las funciones lógicas y aritméticas de transformación.* Es el conjunto de operadores del sistema digital generalmente reunidos bajo la forma de un único dispositivo llamado *unidad aritmetico-lógica* (*ALU*). Estos operadores no son siempre necesarios: los ejemplos más conocidos de sistemas digitales sin *ALU* son la máquina de Turing y la máquina de decisión binaria. En estos dos casos la única operación realizada por el propio sistema es la transferencia simple (escritura) de datos a una posición de memoria. Estos son ejemplos extremos de la simulación por programación de las funciones realizadas

normalmente por hardware. La frontera entre software y hardware es arbitraria y es en el momento de la selección de los operadores del sistema digital cuando se delimita la línea divisoria.

4) *Un conjunto de caminos de datos* (los buses de datos). Que son los encargados de unir las diferentes posiciones de memoria (los operandos) y los operadores.

5) *Un elemento o unidad de control.* Que permite ejecutar las operaciones en el orden preestablecido por el programa. En efecto, el camino de datos que une los operandos de una μ_{op} no debe estar siempre abierto: la transferencia se debe realizar únicamente en los momentos indicados por el programa. Es entonces necesario añadir una serie de elementos en la Figura 5.5: un conjunto de señales de control llamadas *microórdenes* (μ_{ord}) que abren o cierran el camino de datos permitiendo o impidiendo la ejecución de la μ_{op} (ver Figura 5.6).

   El término *microoperación* describe así una acción elemental realizada sobre la ruta de datos, como por ejemplo transferir un dato de un registro fuente a un registro destino. Esta acción elemental sucede en un instante de tiempo particular. *Cada μ_{op} puede implicar la activación de una o más microórdenes.* Desde este punto de vista una *microorden* es una señal binaria generada por la unidad de control. Las microórdenes representan pues acciones atómicas e indivisibles que se generan en la unidad de control para que se lleven a cabo los procesos necesarios en la ruta de datos.

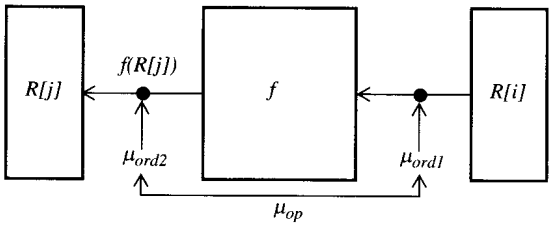


Figura 5.6: μ_{op} de transferencia entre registros controlada por dos μ_{ord}'s

La Figura 5.7 muestra el esquema de un sistema digital con los 5 componentes que se acaban de describir.

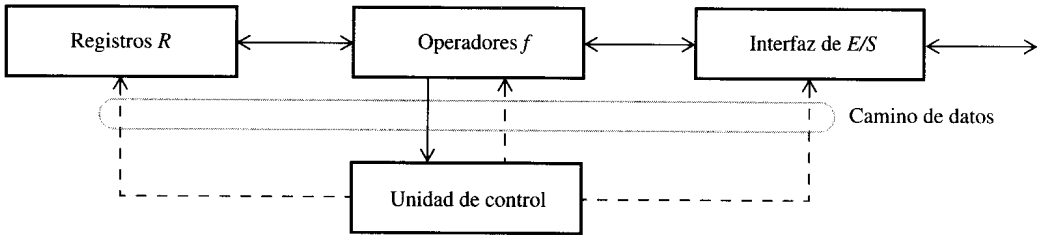


Figura 5.7: Esquema general de un sistema digital

5.3.2 Puntos de control

Los puntos de control permiten la ejecución de las diferentes microórdenes y se emplean para controlar la

selección de los operadores y la carga de los registros. Los puntos de control pueden considerarse como una especie de conmutadores generalizados que estan activados por las microórdenes y que permiten controlar el flujo de información a través de la estructura del sistema digital. Se realizan mediante circuitos cuya naturaleza depende de las características de los dispositivos conectados a los caminos de datos sobre los que actúan. Los ejemplos siguientes muestran algunas posibilidades:

1) El punto de control c_2, que controla el almacenamiento de datos en el registro A, se puede realizar conectando dicha línea a la entrada "carga en paralelo" (CP) del registro. Lo mismo se puede decir de los puntos de control c_0, que controla el desplazamiento a derechas del registro A (DD) y c_1 que permite poner a 0 su contenido (Z), tal como se muestra en la Figura 5.8.

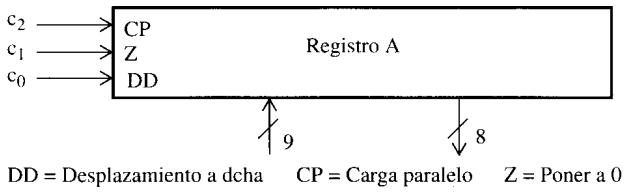


Figura 5.8: Puntos de control realizados directamente activando señales de control

2) Las señales A y B, a través de los puntos de control c_0 y c_1 respectivamente vuelcan su contenido en el bus de salida. Para estos puntos de control no se pueden emplear puertas lógicas convencionales, ya que aunque una de las dos puertas no estuviese seleccionada su salida podría forzar los niveles lógicos que deben aparecer sobre el bus. Esto se pone de manifiesto en la Figura 5.9.

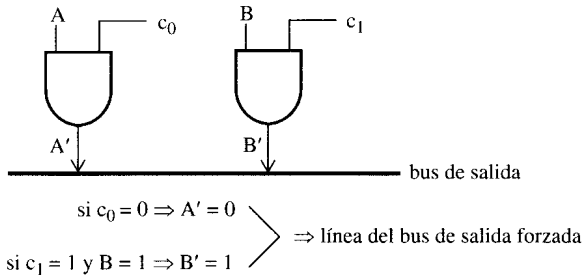


Figura 5.9: Salida forzada cuando se emplean puertas lógicas convencionales

Para evitar estos problemas en la conexión al bus se utilizan drivers triestado (ver Figura 5.10). En la sección 1-4 se consideró el problema de la conexión a un bus mediante lógica triestado.

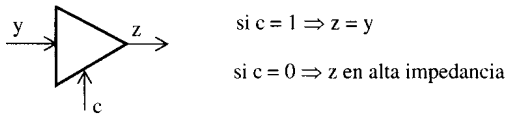


Figura 5.10: Driver triestado

En la Figura 5.11 se muestra como se realizan los puntos de control c_0 y c_1 utilizando drivers triestado.

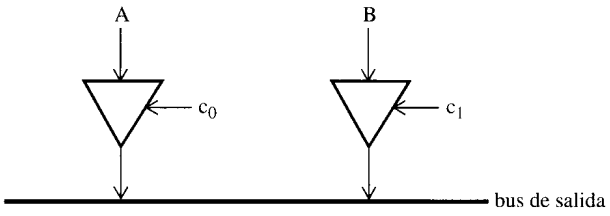


Figura 5.11: Puntos de control en la conexión de A y B con el bus de salida

3) En la Figura 5.12 se muestra un ejemplo de un punto de control c realizado mediante una puerta de palabras (ver sección A-1 en el Apéndice A).

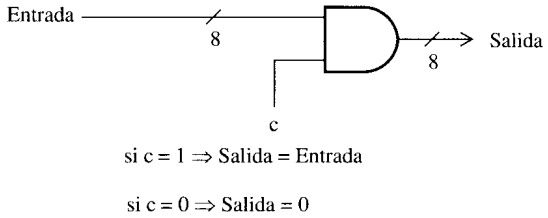


Figura 5.12: Realización de un punto de control mediante una puerta de palabras

5.3.3 Modelo de Glushkov

El modelo de Glushkov es un modo de representación de los sistemas digitales que utiliza un nivel de abstracción superior al que se ha visto en el apartado anterior. En efecto, este modelo descompone un sistema digital en dos partes únicamente:

1) *Una unidad de procesamiento o ruta de datos.* Que comprende los elementos de memoria, la interfaz con el exterior, el conjunto de operadores aritméticos y lógicos (recursos de cálculo) y los caminos de transferencia de datos. Esta unidad realiza las operaciones sobre los datos y asegura su memorización.

2) *Una unidad de control.* Que recibe de la unidad de procesamiento una indicación sobre el estado de las operaciones y en función de estos valores y del estado del programa, controla la secuencia de ejecución de las microoperaciones, comienza su ejecución enviando las microórdenes adecuadas y determina cuál es el paso futuro del programa. Toda manipulación de datos se controla por las microórdenes generadas por esta unidad. Esta unidad es por lo tanto una *máquina secuencial*, que es la realización en hardware del algoritmo del sistema. La unidad de control produce así secuencias de *señales de control* que gobiernan los cálculos realizados en la unidad de procesamiento.

En la Figura 5.13 se muestra el esquema de un procesador de acuerdo con el modelo de Glushkov.

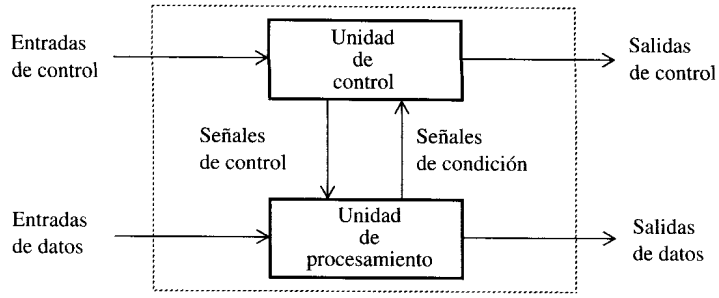


Figura 5.13: Modelo de Glushkov de un sistema digital

Las entradas y salidas del sistema constan de *entradas y salidas de datos* y *entradas y salidas de control.* Las unidades de control y de procesamiento se comunican por medio de *señales de control* y de *condición o estado.* Las condiciones generadas por la unidad de procesamiento (señales de condición) se utilizan por la unidad de control para producir un secuenciamiento de las señales de control. Las unidades de control y de procesamiento se definen especificando los siguientes elementos:

1) Un conjunto de *componentes*

2) La *estructura de interconexión* entre los componentes

3) La *función* realizada por los componentes interconectados

De forma más concreta la estructura de la unidad de procesamiento o ruta de datos (ver Figura 5.14) incluye:

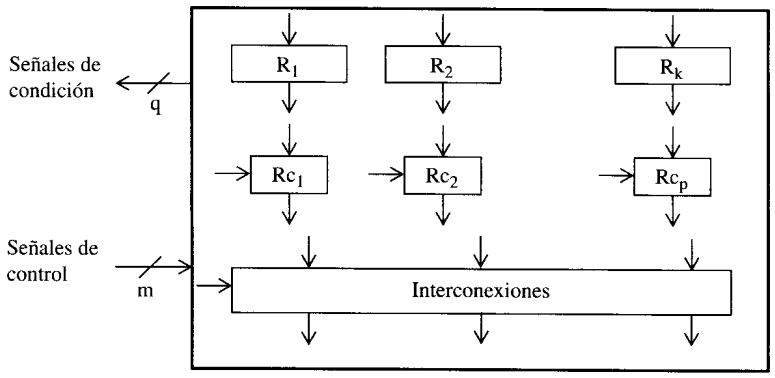


Figura 5.14: Estructura general de la unidad de procesamiento

a) *Un conjunto de registros* $\{R_1, R_2, ..., R_k\}$ que almacenan los datos procesados (datos iniciales, cálculos intermedios y resultados finales).

b) *Un conjunto de primitivas o recursos de cálculo* $\{Rc_1, Rc_2, ..., Rc_p\}$ que realizan el tratamiento sobre los datos. Son el conjunto de operadores aritméticos y lógicos que contiene la unidad de procesamiento.

c) *Una red de interconexión* capaz de conectar los registros con los recursos de cálculo.

d) Un conjunto de entradas de control denominadas *puntos de control.*

e) Un conjunto de salidas que constituyen las *señales de condición o estado.*

La función que realiza un componente (recurso de cálculo, registro o red de interconexión) se determina por las señales de control generadas en la unidad de control que están presentes en los puntos de control.

Así pues, los recursos de cálculo, los registros de almacenamiento y la red de interconexiones se controlan externamente a través de la unidad de control que es un *sistema o máquina secuencial* (a veces se denomina *autómata operacional*). Su función de transición especifica el secuenciamiento de las microórdenes y su función de salida determina las señales de control que activan las microórdenes ejecutadas por la unidad de procesamiento.

Las entradas de la unidad de procesamiento (provenientes de la unidad de control) vienen codificadas por medio de *m señales de control* c_1, c_2, ..., c_m, cada una de las cuales define el estado de una línea de control. Ejemplos de líneas de control son: activar la carga de un determinado registro, programar la función que realiza un recurso de cálculo y las líneas de control de la red de interconexión.

Las salidas de la unidad de procesamiento (que las recibe como entradas la unidad de control) se codifican como *q señales de condición o estado* s_1, s_2, ..., s_q. Estas señales proporcionan información a la unidad de control acerca del estado actual de la unidad de procesamiento (ver Figura 5.15).

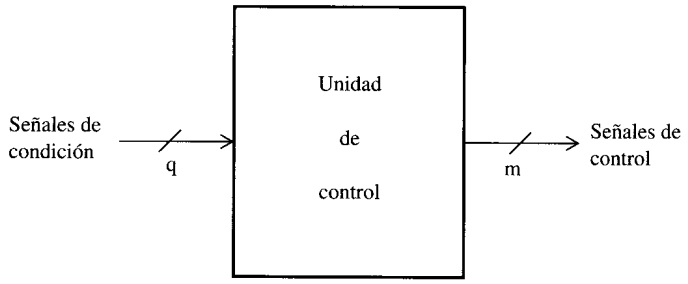


Figura 5.15: Estructura de entrada-salida de la unidad de control

El modelo de Glushkov que se acaba de presentar ayuda a clasificar los procesadores. En efecto, como un procesador es un dispositivo que realiza una secuencia de operaciones sobre sus datos, su clasificación está determinada por el tipo de secuencias que realiza, es decir por la *estructura de su unidad de control* y por el tipo de operaciones que ejecuta y el tipo de datos sobre las cuales actúa, es decir por la *estructura de su unidad de procesamiento o ruta de datos.* Este modelo proporciona además una nueva prueba de la analogía hardware-software establecida por la famosa ecuación de Wirth:

$$programas = algoritmos + estructura\ de\ datos$$

que ahora se reemplaza por:

$$procesadores = unidad\ de\ control + unidad\ de\ procesamiento$$

Sección 5-4

Diagramas de máquinas de estados algorítmicas (ASM)

El *diagrama de máquina de estados algorítmica* o *diagrama ASM* (del inglés Algorithmic State Machines) es una forma gráfica, de especificar el modelo de un sistema digital lo constituye. El ASM se asemeja a un organigrama convencional, pero se interpreta de forma algo diferente. En un organigrama clásico se describen los pasos y los caminos de decisión de un algoritmo sin preocuparse de su relación con el tiempo. Por el contrario el diagrama ASM se caracteriza por el hecho de que describe una secuencia de sucesos así como la relación temporal entre los estados de la unidad de control y las acciones que ocurren en los estados como respuesta a los distintos pulsos de reloj.

Un diagrama ASM se define mediante cuatro componentes básicos (ver Figura 5.16): el bloque de estado, el bloque de decisión, el bloque de condición y el bloque ASM.

- *Bloque de estado.* Cada estado en un diagrama ASM se indica mediante un bloque de estado, que es un rectángulo que contiene las operaciones de transferencia de registro y las señales de salida que se activan mientras la unidad de control se encuentra en ese estado. Cada bloque de estado posee un nombre simbólico que se coloca en el ángulo superior izquierdo de su rectángulo. El código binario para dicho estado, si se ha asignado, se situa en el ángulo superior derecho del rectángulo.

  En la Figura 5.16 se muestra un ejemplo específico de un bloque de estado. El estado tiene el nombre simbólico *Inicio* y el código binario asignado es 000. Dentro del bloque se encuentra la operación de transferencia de registro $R \leftarrow 0$ y la salida *Apagado*. La transferencia de registro indica que el registro R se inicializa a 0 en cualquier pulso de reloj que ocurra con el control en el estado *Inicio*. *Apagado* denota que la señal de salida que tiene ese nombre vale 1 durante todo el tiempo que el control permanezca en el estado *Inicio*. *Apagado* vale 1 en cualquier bloque de estado en el cual aparezca y 0 en todos los otros bloques de estado.

- *Bloque de decisión.* Este bloque describe la condición bajo la que se llevarán a efecto determinadas acciones en la unidad de tratamiento o ruta de datos y se seleccionará el próximo estado. Estas condiciones pueden referirse tanto a las entradas de control externas como a las señales de estado. El bloque de decisión que tiene forma de rombo tiene dos salidas posibles, una se escoge cuando la expresión booleana que expresa la condición indicada dentro del bloque es verdadera (*Dato* = 1 en la Figura 5.16) y la otra cuando es falsa (*Dato* = 0).

- *Bloque de condición.* Este bloque es característico de los diagramas ASM. Describe las asignaciones de variables o salidas que se realizan cuando se cumplen las condiciones especificadas por uno o más bloques de decisión. Los bordes redondeados del bloque de salida condicional permiten distinguirlo del bloque de estado.

- *Bloque ASM.* Es una estructura compuesta que consta de un bloque de estado y un conjunto de bloques de decisión y de condición conectados en serie y/o paralelo. Tiene una única entrada aunque puede tener cualquier número de salidas.

  En el ejemplo de la Figura 5.16 se observa que si se está en el estado *Inicio*, el registro *Final* se inicializa a 0 y también lo hace el registro *Dato* pero sólo si la señal *Activa* es igual a 1.

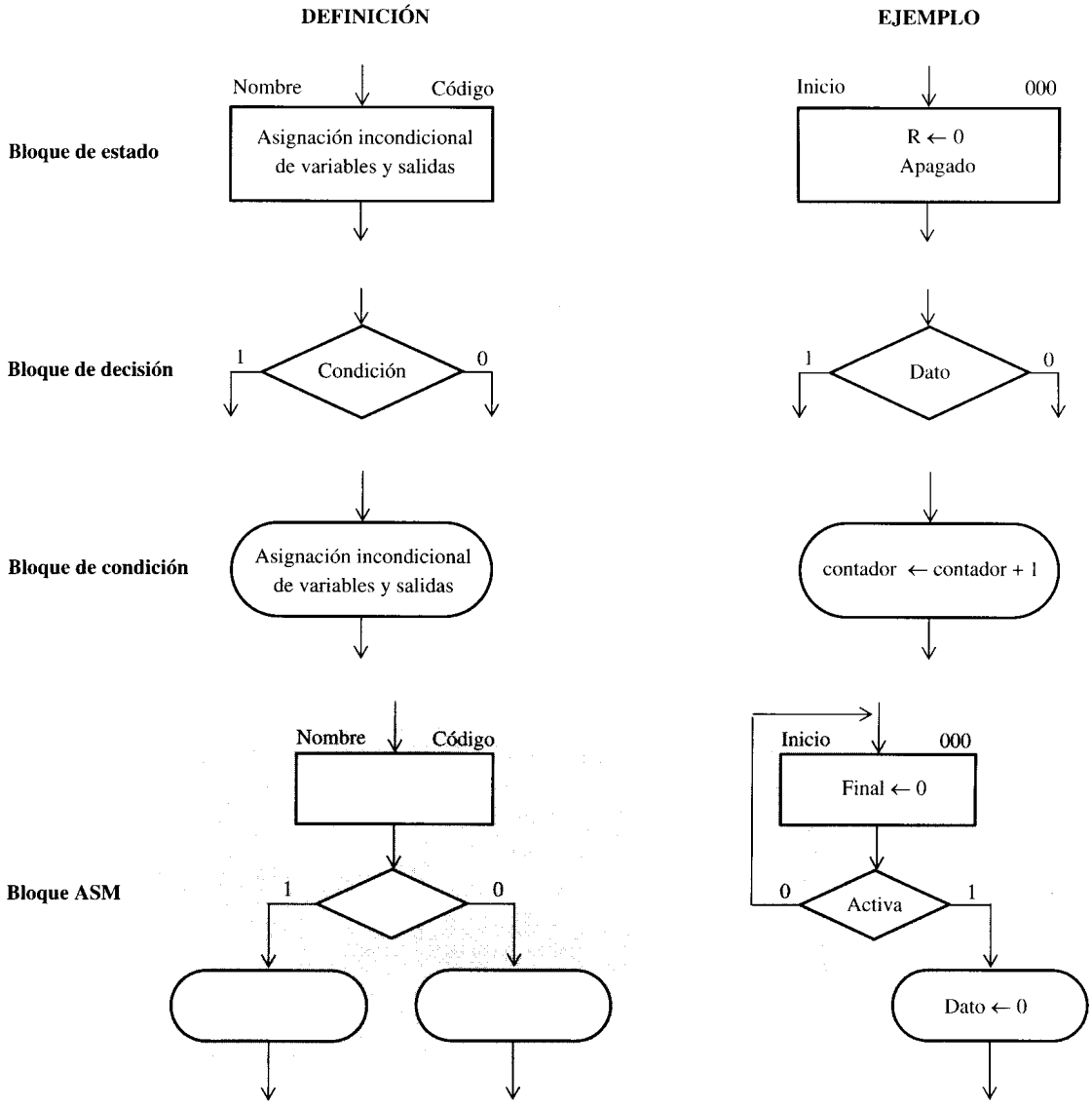


Figura 5.16: Elementos de un diagrama ASM

A partir de estos bloques básicos se puede definir que es un diagrama ASM:

Definición

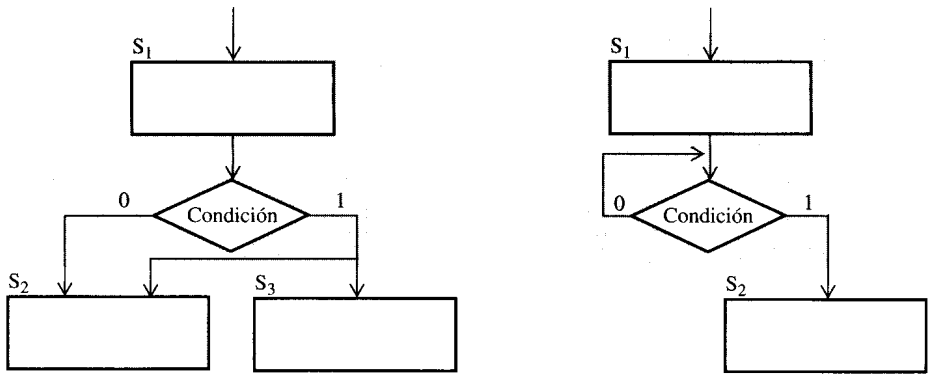
Un *diagrama ASM* está compuesto por uno o más bloques ASM conectados y dispuestos de tal forma que cada salida de un bloque es la entrada de un único bloque de estado. ♦

Cada uno de los bloques ASM describe las acciones que se realizan en un estado, de forma que se pueden considerar equivalentes a cada una de las filas de una tabla de estados de una máquina secuencial (ver Apéndice B). La única diferencia está en la manera en que se expresan las condiciones. En un bloque ASM se combinan bloques de decisión y condición. Cuando se describe el modelo de un sistema digital utilizando un diagrama ASM, hay que respetar las dos reglas siguientes:

1) El diagrama debe definir un único próximo estado para cada uno de los estados presentes y conjunto de condiciones.

2) Todo camino definido por el conjunto de bloques de condición debe llevar necesariamente a otro estado.

En la Figura 5.17 se muestran dos ejemplos de diagramas ASM que son incorrectos porque no satisfacen alguna de las dos reglas anteriores. El diagrama de la Figura 5.17a no satisface la primera regla, ya que cuando *Condición* es cierta, tanto S_2 como S_3 están especificados como próximos estados. De forma análoga el diagrama de la Figura 5.17b no cumple la segunda regla ya que el camino definido cuando *Condición* es falsa forma un bucle que no lleva a ningún estado.



a) Estado siguiente no definido **b) Camino de salida no definido**

Figura 5.17: Diagramas ASM incorrectos

Sección 5-5

Ejemplo de diseño: multiplicador binario

En esta sección se va a desarrollar de forma detallada, a modo de ejemplo ilustrativo, la implementación hardware de un multiplicador binario de dos números sin signo de n bits cada uno utilizando el modelo de Glushkov y los diagramas ASM vistos en la sección anterior. Para la multiplicación se emplea el algoritmo de *lápiz y papel mejorado* propuesto en el tema 4 (ver apartado 4.4.2).

5.5.1 Multiplicador binario

En el diseño del multiplicador se van a emplear únicamente circuitos que se han considerado con anterioridad. El algoritmo que se utiliza es el de lápiz y papel mejorado. Para analizar y recordar el funcionamiento de este algoritmo se considera en primer lugar la multiplicación binaria de una pareja de números de 4 bits. Sea el multiplicando M = 1101 y el multiplicador m = 1011 tal como se muestra en la Figura 5.18a. Para realizar la multiplicación con el algoritmo de lápiz y papel se examinan de forma sucesiva los bits del multiplicador comenzando por el menos significativo. Si el bit del multiplicador es 1, se copia el contenido del multiplicando para efectuar la suma posteriormente, en caso contrario se copian 0's. Los números copiados en líneas sucesivas se desplazan una posición hacia la izquierda de los números previamente copiados, para situarse en el alineamiento correcto con el bit del multiplicador que está siendo procesado. Finalmente, todos los números copiados se suman y el resultado es el producto. Si se multiplican dos números de n bits el resultado puede tener hasta $2n$ bits. Cuando el procedimiento de multiplicación se implementa con hardware digital, es conveniente cambiar ligeramente el proceso descrito.

1) En lugar de disponer de un circuito digital que sume simultáneamente n números binarios, es menos costoso tener un circuito que sume sólo dos números. Así que cada vez que se determine que hay que sumar una copia del multiplicando o de 0's se hace de forma inmediata para formar un *producto parcial*. Este producto parcial se almacena en un registro antes de realizar la operación de desplazamiento.

2) En lugar de desplazar las copias del multiplicando hacia la izquierda, el producto parcial que se está formando se desplaza hacia la derecha. Esto deja al producto parcial y a la copia del multiplicando en la misma posición relativa que cuando se desplazaba hacia la izquierda el multiplicando. Pero más importante aún es que sólo se necesita un sumador de n bits en lugar de uno de $2n$ bits. La suma siempre tiene lugar en las mismas n posiciones, en lugar de moverse cada vez un bit hacia la izquierda.

3) Cuando el bit correspondiente en el multiplicador es 0, no hay necesidad de sumar todos los ceros al producto parcial porque esto no altera el valor del resultado.

En la Figura 5.18b se repite el ejemplo de multiplicación binaria con estos cambios. Obsérvese que el producto parcial inicial es 0. Cada vez que el bit del multiplicador que está siendo procesado es 1, se efectúa una suma del multiplicando seguida por un desplazamiento hacia la derecha. Cada vez que el bit del multiplicador es 0, sólo se realiza un desplazamiento hacia la derecha. Una de estas dos acciones se ejecuta para cada bit del multiplicador, de forma que en este ejemplo ocurren cuatro acciones de este tipo.

Conviene advertir que en el proceso de suma puede ocurrir un rebose (en el ejemplo de la Figura 5.18b se muestra en negrita). Sin embargo este arrastre no representa ningún problema puesto que inmediatamente ocurre un desplazamiento hacia la derecha que lleva al bit extra del producto parcial a la posición del bit más significativo.

```
 13          1101    multiplicando
 11          1011    multiplicador
             1101
            1101
           0000
          1101
143     10001111    producto
```

a) Algoritmo de lápiz y papel

```
13   1101        multiplicando
11   1011        multiplicador
     0000        producto parcial inicial
     1101        sumar el multiplicando porque el bit del multiplicador es 1
     1101        producto parcial después de sumar y antes de desplazar
     01101       producto parcial después de desplazar
     1101        sumar el multiplicando porque el bit del multiplicador es 1
     100111      producto parcial después de sumar y antes de desplazar
     010011      producto parcial después de desplazar
     1101        sumar el multiplicando porque el bit del multiplicador es 1
     10001111    producto parcial después de sumar y antes de desplazar
143  10001111    producto después del desplazamiento final
```

b) Algoritmo de lápiz y papel mejorado

Figura 5.18: Ejemplo manual de multiplicación binaria

En la Figura 5.19 se muestra el algoritmo básico de multiplicación de lápiz y papel mejorado que se desea implementar. Se ha utilizado la notación || para definir un registro compuesto constituido a partir de otros registros. Así por ejemplo $C||A||m$ representa un único registro obtenido al combinar en ese orden los registros C, A y m (el bit más significativo está en C y el menos significativo es el bit más a la derecha del registro m).

El símbolo >> especifica un desplazamiento del dato hacia la derecha, indicándose a continuación del símbolo el número de bits de desplazamiento (en este caso 1). El bit menos significativo del registro m se representa por $m[0]$. Al sumar el contenido de los registros A y M se puede producir un arrastre en la operación, dicho arrastre se representa por c_a.

```
1: M ← Entrada;
2: m ← Entrada;
3: A ← 0;
4: C ← 0;
5: P ← n - 1;
   for P := n - 1 downto 0 do
6:    if m[0] = 1 then A ← A + M, C ← c_a;
7:    C ← 0, C || A || m ← C || A || m >> 1;
8:    P ← P - 1;
   end for
```

Figura 5.19: Algoritmo de multiplicación de lapiz y papel mejorado

5.5.2 Unidad de procesamiento o ruta de datos del multiplicador

Un análisis del algoritmo de la Figura 5.19 muestra que para la implementación de la unidad de procesamiento correspondiente son necesarios los siguientes recursos de cálculo:

1) Un registro M de n bits, para almacenar el multiplicando, con capacidad de carga en paralelo.
2) Un registro m de n bits, donde se almacena inicialmente el multiplicador y al final del algoritmo los n bits menos significativos del producto, con capacidad de carga en paralelo y de desplazamiento hacia la derecha.
3) Un registro A de n bits, donde se almacena al final del algoritmo los n bits más signifcativos del producto, con capacidad de carga en paralelo (que acepta como entrada la salida del sumador en la operación $A \leftarrow A + M$) y de desplazamiento hacia la derecha.
4) Un sumador paralelo de dos números de n bits.
5) Un contador P con capacidad de carga en paralelo y posibilidad de contar hacia abajo (decrementar). El contador P requiere $\lceil log_2 n \rceil$ bits para poder contar el procesamiento de los n bits del multiplicador.
6) Un circuito de detección de cero con $\lceil log_2 n \rceil$ entradas y una salida.
7) Un elemento de memoria C que puede ponerse a 0 o cargarse con el valor c_a que es el arrastre o rebose de salida del sumador paralelo.

En la Figura 5.20 se muestra el diagrama de bloques de la unidad de procesamiento o ruta de datos que se acaba de describir.

5.5.3 Diagrama ASM del multiplicador

En la Figura 5.21a se representa un diagrama ASM que describe la secuencia de operaciones del multiplicador binario. Inicialmente el multiplicando está almacenado en el registro M y el multiplicador en el registro m. Se supone que de la carga de estos dos registros no se encarga la unidad de control del multiplicador. Mientras el diagrama ASM se encuentra en el estado S_0 y la señal I (que indica el inicio de la operación) vale 0 no ocurre ninguna acción y el sistema permanece en ese estado. La operación de

multiplicación comienza cuando I se hace igual a 1. Cuando eso sucede, el diagrama se transfiere del estado S_0 al estado S_1, los registros A y C se inicializan a 0 y el contador P se carga con la constante n-1. En el estado S_1 se efectúa una decisión que se basa en el valor de $m[0]$ (señal de condición s_0) que es el bit menos significativo del registro m. Si s_0 es 1, se suman los contenidos de los registros A y M y el resultado se transfiere otra vez al registro A con el posible bit de arrastre de la suma c_a almacenado en el elemento de memoria C. Si s_0 es 0, no se modifican los contenidos de A y C. En ambos casos, el próximo estado es S_2.

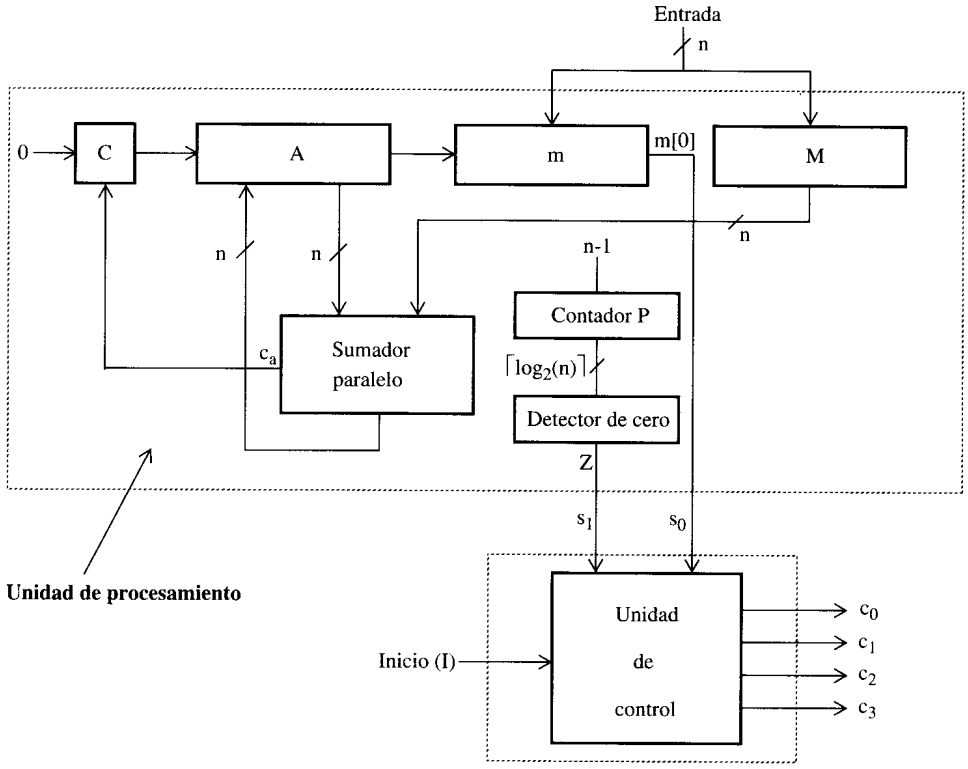


Figura 5.20: Diagrama de bloques del multiplicador binario con el algoritmo de lapiz y papel mejorado

En el estado S_2, se realiza un desplazamiento hacia la derecha del contenido combinado de C, A y m. Este desplazamiento consiste en el siguiente conjunto de transferencias simultáneas de registros:

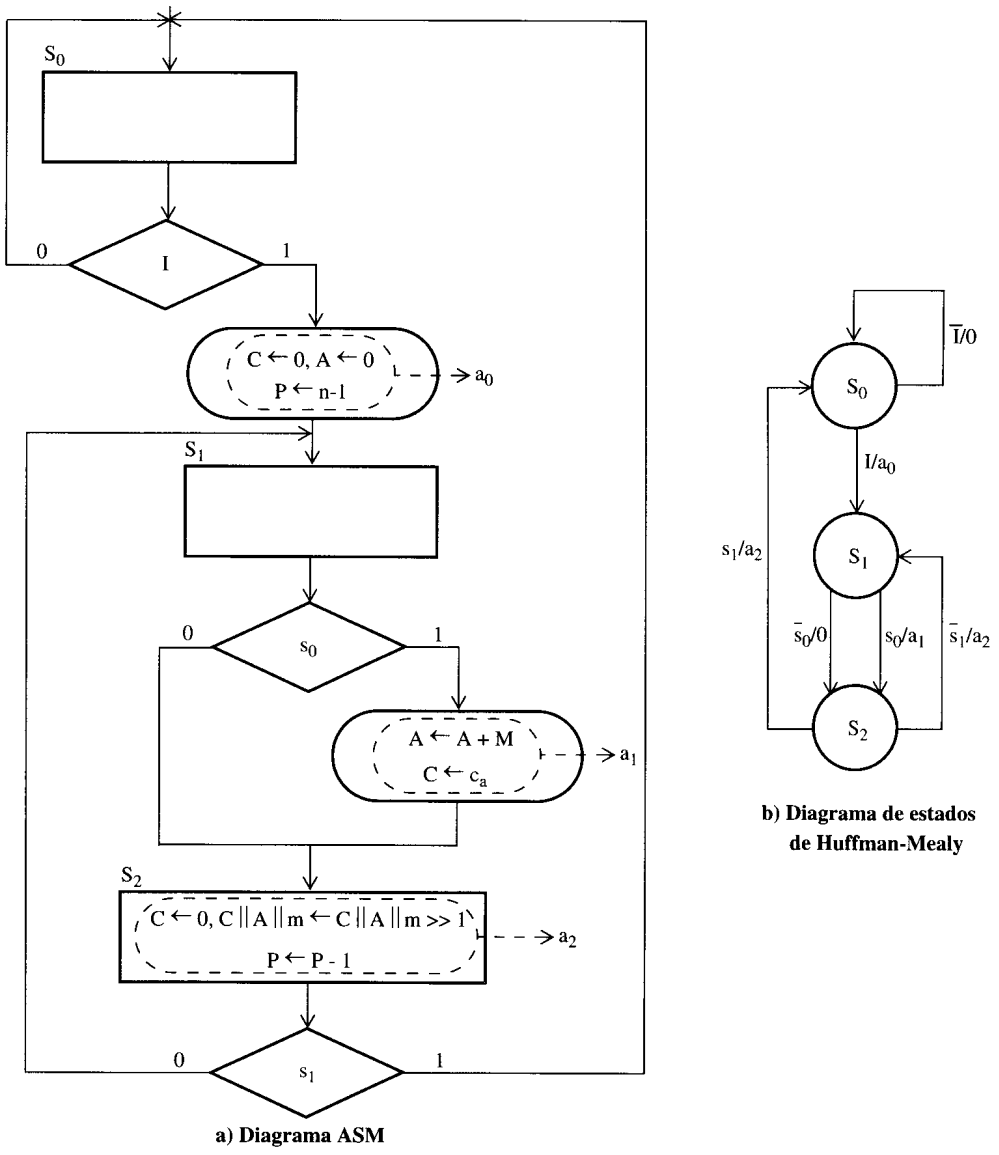
$$C \leftarrow 0, A[n-1] \leftarrow C, A[n-2] \leftarrow A[n-1], ..., A[0] \leftarrow A[1], m[n-1] \leftarrow A[0], m[n-2] \leftarrow m[n-1], ..., m[1] \leftarrow m[0]$$

con la notación introducida en el apartado anterior, esta operación simultánea de transferencia entre registros se puede expresar como:

$$C \leftarrow 0, C \,||\, A \,||\, m \leftarrow C \,||\, A \,||\, m >> 1$$

Conviene recordar que el bit más a la izquierda del resultado de la operación de desplazamiento hacia la derecha, que es el contenido del registro C, toma el valor 0. Sin embargo el registro C se inicializa también a

0 en la línea 4 del algoritmo (ver Figura 5.19) o lo que es equivalente en el bloque de condición asociado al estado S_0 en el diagrama ASM (ver Figura 5.21a).



a) Diagrama ASM

b) Diagrama de estados de Huffman-Mealy

Figura 5.21: Diagrama ASM del multiplicador binario

El contador P se decrementa en el estado S_2. El valor de P se comprueba después de la formación de cada producto parcial, pero antes de dicha comprobación se decrementa su valor. Durante las n-1 veces que se comprueba que el contenido de P es distinto de cero, el bit de estado Z (señal de condición s_1) es 0 y se ejecuta el bucle que contiene los estados S_1 y S_2. La siguiente vez que se examina P su contenido es cero, de

forma que s_1 pasa a valer 1. Esto indica que se ha finalizado la multiplicación y el diagrama ASM retorna al estado S_0. El resultado final se encuentra entonces disponible en el registro combinado $A \| m$, con A almacenando los n bits más significativos y en m los n bits menos significativos del producto.

Es interesante advertir en este punto que cuando un diagrama ASM posee salidas condicionales (bloques de condición) el modelo se corresponde con un modelo de máquina secuencial tipo Huffman-Mealy (ver el apartado B.1.2 en el Apéndice B sobre la estructura canónica de una máquina secuencial). En la Figura 5.21b se muestra el diagrama de estados que se asocia al diagrama ASM del multiplicador binario de la Figura 5.21a. Las acciones que se realizan (que son las salidas del circuito secuencial) están asociadas a las transiciones entre estados. En el diagrama de estados estas acciones se representan por a_0, a_1 y a_2 respectivamente.

Sin embargo si se desea utilizar un modelo de máquina secuencial tipo Moore donde las salidas en un instante de tiempo dado dependen sólo del estado en el que se encuentra el sistema, *es preciso sustituir en el diagrama ASM los bloques de condición por bloques de estado*. Esto significa que se requieren más estados en el diagrama ASM para implementar un determinado algoritmo. En la Figura 5.22a se muestra un diagrama ASM para el multiplicador binario desarrollado con la restricción de que el sistema no contiene bloques de salida condicional y en la Figura 5.22b el diagrama de estados que le corresponde. Las salidas que están asociadas a los estados S_{00}, S_{10} y S_2 se representan en el diagrama de estados por a_0, a_1 y a_2 respectivamente. En los estados S_0 y S_1 que no requieren la activación de ninguna salida se coloca un 0 en el diagrama de estados. Las transiciones entre estados que no van etiquetadas (de S_{00} a S_1 y de S_{10} a S_2) se producen siempre independientemente del valor que tengan las señales de condición o las señales externas.

Comparado con el diagrama ASM de la Figura 5.21a este diagrama tiene dos estados más, los estados S_{00} y S_{10} que han sido colocados donde en el diagrama original se utilizaban bloques de condición. Además de ser un circuito tipo Moore (ver Figura 5.22b) este diagrama ASM sólo posee bloques de decisión simples para determinar el secuenciamiemto entre estados.

5.5.4 Unidad de control con lógica cableada

En el diseño de una unidad de control hay, como ya se ha visto, dos aspectos distintos que hay que considerar:

1) *El control de las microoperaciones*, que se encarga de generar las señales de control.
2) *El secuenciamiento de la unidad de control y de las microoperaciones*, que tiene como objetivo determinar lo que sucede a continuación.

Es posible independizar estas funciones al separar el diagrama ASM original en dos partes:

a) Una *tabla de estados* que define las señales de control en términos de los estados y de las entradas.
b) Un *diagrama ASM simplificado* que representa solamente las transiciones entre estados.

En la Tabla 5.2 se relacionan las señales de control que se necesitan para la unidad de procesamiento o ruta de datos del multiplicador. El procedimiento seguido ha sido examinar todos los registros de la unidad de procesamiento y tabular las microperaciones que se realizan en cada uno de ellos. Se ha tomado en consideración que una misma señal de control pueda activar microoperaciones en más de un registro. Esto parece razonable en este caso ya que se está diseñando una unidad de procesamiento dedicada a realizar una única operación, la multiplicación. No se separan pues todas las señales de control lo que proporcionaría una

generalidad que no se requiere en este caso. La ecuación lógica de cada señal de control (c_i, i = 0, 1, 2 y 3) se obtiene a partir de la posición o posiciones de la microoperación correspondiente en el diagrama ASM.

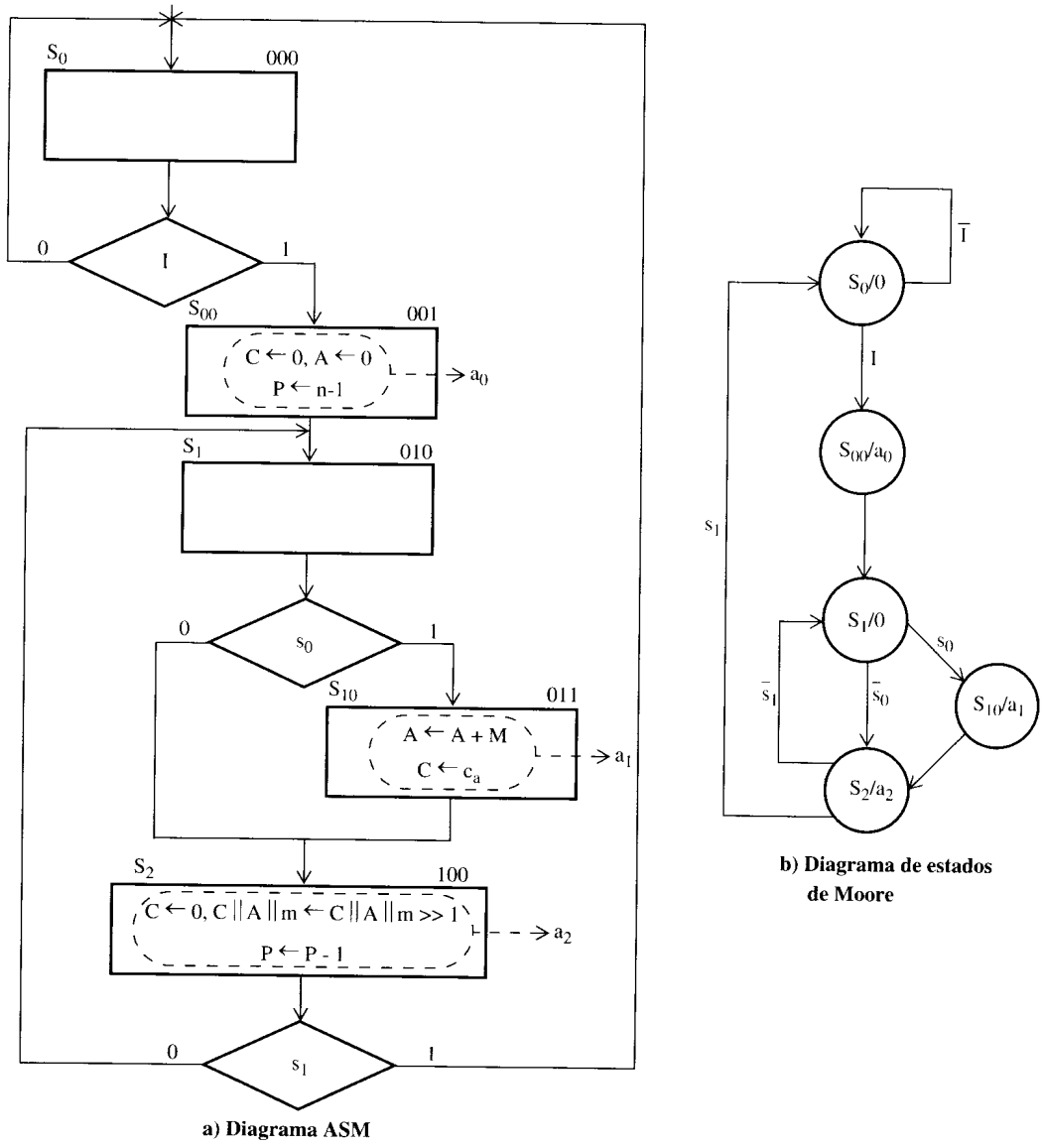


Figura 5.22: Diagrama ASM del multiplicador binario sin bloques de salida condicional

Así por ejemplo, para el registro A en la Tabla 5.2 se muestran tres microoperaciones; *Inicializar* (c_0), *Sumar_Cargar* (c_1) y *Desplazar_Dec* (c_2). Como la operación de poner a cero el registro A ocurre al mismo tiempo que las operaciones de poner a cero también el elemento de memoria C y de cargar el contador P, todas estas microoperaciones se podrían activar por la misma señal de control c_0. Sin embargo el elemento de

memoria C también se inicializa a cero cuando se está en el estado S_2 y por esta razón se elige independizar la señal de control de esta microperación (c_3). Así pues la señal de control de *Inicializar* se utiliza para poner a cero A y cargar P.

En la última columna de la Tabla 5.2 se dan las expresiones boolenas de cuando están activas las diferentes señales de control. Estas expresiones se deducen directamente del diagrama ASM, por ejemplo la activación de la señal de control *Inicializar* (c_0) se produce cuando el sistema está en el estado S_0 y la entrada externa I vale 1 ($c_0 = S_0I$). Obsérvese que en este punto se está tratando el nombre del estado S_0 como una variable booleana. La señal de control *Borrar_C* (c_3) que pone a cero C, se activa cuando se está en el estado S_0 y la entrada externa I vale 1 o cuando se está en el estado S_2 ($c_3 = S_0I + S_2$). Las otras dos señales de control internas del multiplicador *Sumar_Cargar* (c_1) y *Desplazar_Dec* (c_2) se definen de forma análoga. Las dos señales *Cargar_M* y *Cargar_m*, cargan el multiplicando y el multiplicador desde fuera del multiplicador y por lo tanto no se consideran de forma explícita en el resto del diseño.

Si en el diagrama ASM de la Figura 5.21a se elimina la información sobre las microperaciones queda sólo representado el secuenciamiento entre estados. En la Figura 5.23a se muestra este diagrama ASM modificado en el que tampoco aparecen los bloques de condición.

También se puede quitar cualquier bloque de decisión si no influye sobre cuál es el próximo estado al que se transfiere el sistema. En particular, en la Figura 5.21a el bloque de decisión s_0 afecta sólo a un bloque de condición. Una vez se ha eliminado este bloque de condición, las dos salidas del bloque de decisión s_0 claramente van al mismo estado, de forma que no afecta al próximo estado y se puede eliminar.

Componente	Microoperación	Señal de control	Ecuación lógica
Registro A	A ← 0 A ← A + M C || A || m ← C || A || m >> 1	Inicializar Sumar_Cargar Desplazar_Dec	$c_0 = S_0I$ $c_1 = S_1s_0$ $c_2 = S_2$
Registro M	M ← Entrada	Cargar_M	
Elemento de memoria C	C ← 0 C ← c_a	Borrar_C Sumar_Cargar	$c_3 = S_0I + S_2$ $c_1 = S_1s_0$
Registro m	m ← Entrada C || A || m ← C || A || m >> 1	Cargar_m Desplazar_Dec	 $c_2 = S_2$
Contador P	P ← n - 1 P ← P - 1	Inicializar Desplazar_Dec	$c_0 = S_0I$ $c_2 = S_2$

Tabla 5.2: Tabla de estados para la unidad de control del multiplicador binario

A partir de este diagrama ASM modificado, se puede diseñar la parte de secuenciamiento de la unidad de control (es decir la parte que representa la conducta del próximo estado). La Figura 5.23b corresponde al diagrama de estados de un circuito secuencial en el que no se han especificado las salidas. Se observa que existe una correspondencia clara entre ambos diagramas por lo que se puede utilizar directamente el diagrama ASM modificado para formar la tabla de estados del circuito secuencial de la unidad de control del multiplicador.

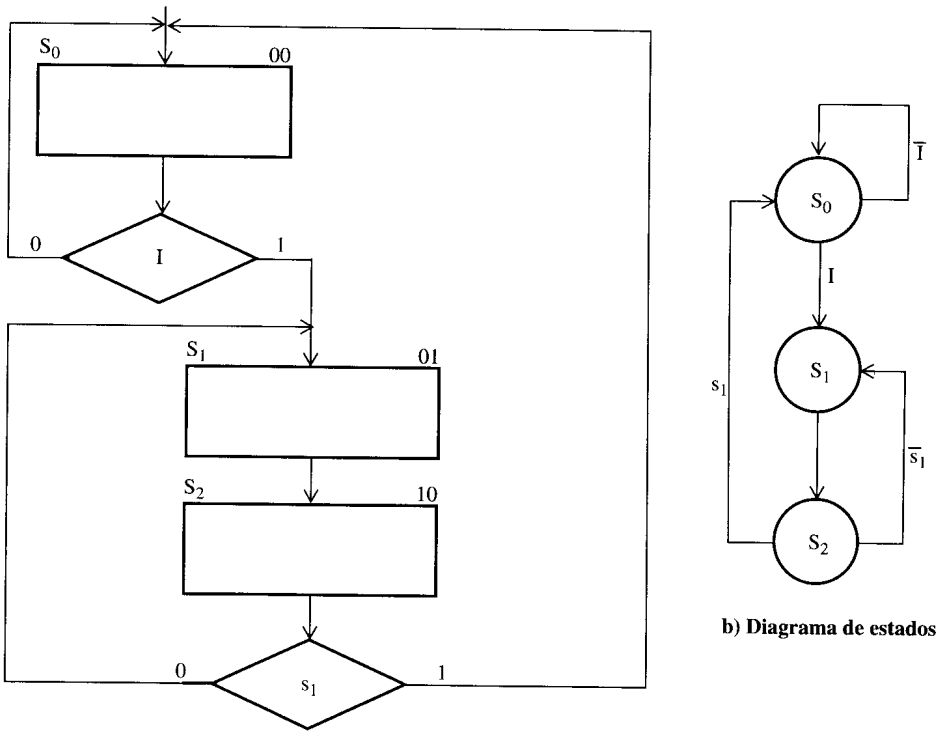


Figura 5.23: Secuenciamiento del multiplicador binario

5.5.5 Unidad de control diseñada con elementos de memoria tipo D

Únicamente se van a considerar unidades de control que se modelan como circuitos secuenciales síncronos. Es decir, las transiciones entre los estados en la unidad de control son síncronas con el reloj del sistema. Existen varios métodos de realizar la unidad de control que van desde, estructuras fijas denominadas de *lógica cableada* sintetizadas con elementos de memorias y puertas lógicas, a estructuras muy flexibles y fácilmente modificables de tipo *programable*.

En este apartado y en los siguientes se analizan diferentes realizaciones de la unidad de control del multiplicador binario que se ha venido tratando. En la Figura 5.24 se muestra el diagrama de bloques de la unidad de control y el significado de las señales de control y de condición asociadas.

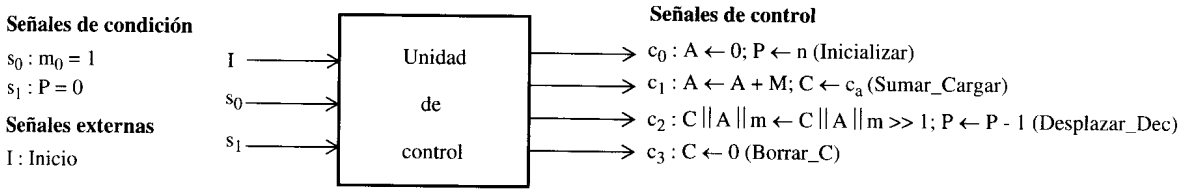


Figura 5.24: Diagrama de bloques de la unidad de control del multiplicador binario

En la Tabla 5.2 se han deducido las siguientes expresiones lógicas de las señales de control c_i (i = 0, 1, 2 y 3) en función de los estados S_i (i = 0, 1 y 2) de la unidad de control y de las señales de condición y de las señales externas

$$\begin{aligned} c_0 &= S_0\, I \\ c_1 &= S_1\, s_0 \\ c_2 &= S_2 \\ c_3 &= S_0\, I + S_2 = c_0 + c_2 \end{aligned} \qquad (5.3)$$

Como las señales de control (que son las salidas del circuito secuencial) son función de las señales de condición y de las señales externas además del estado, el modelo del circuito secuencial asociado es del tipo Huffman-Mealy. Conviene advertir que en este caso el número de estados que se necesita para implementar la unidad de control es de tres (ver Figura 5.23b) en lugar de los cinco que se precisan con un modelo de Moore (ver Figura 5.22b). Para completar el diseño queda únicamente por sintetizar las transiciones entre estados que representan la conducta secuencial de la unidad de control. En la Tabla 5.3 se representa la tabla de estados para el modelo de Huffman-Mealy (deducida a partir de la Figura 5.21b). En la Tabla 5.4 se muestran la asignación de estados elegida (se requieren dos elementos de memoria Q_1 y Q_0) y la tabla de transición de estados y de salida que corresponden a la tabla de estados de la Tabla 5.3. Se observa que la tabla de transición de estados depende de las señales I y s_1 pero es independiente de la señal de condición s_0 que sólo afecta a la señal de control c_1.

Estado presente	Próximo estado			
	$\bar{I}\,\bar{s}_1$	$\bar{I}\,s_1$	$I\,\bar{s}_1$	$I\,s_1$
S_0	S_0	S_0	S_1	S_1
S_1	S_2	S_2	S_2	S_2
S_2	S_1	S_0	S_1	S_0

Tabla 5.3: Tabla de estados del modelo de Huffman-Mealy

$Q_1\,Q_0$
$S_0 \rightarrow$ 0 0
$S_1 \rightarrow$ 0 1
$S_2 \rightarrow$ 1 0

a) Asignación de estados

$Q_1\,Q_0$	Próximo estado				Salida ($c_0\,c_1\,c_2\,c_3$)							
	$\bar{I}\,\bar{s}_1$	$\bar{I}\,s_1$	$I\,\bar{s}_1$	$I\,s_1$	$\bar{I}\,\bar{s}_1\,\bar{s}_0$	$\bar{I}\,\bar{s}_1\,s_0$	$\bar{I}\,s_1\,\bar{s}_0$	$\bar{I}\,s_1\,s_0$	$I\,\bar{s}_1\,\bar{s}_0$	$I\,\bar{s}_1\,s_0$	$I\,s_1\,\bar{s}_0$	$I\,s_1\,s_0$
0 0	0 0	0 0	0 1	0 1	0000	0000	0000	0000	1001	1001	1001	1001
0 1	1 0	1 0	1 0	1 0	0000	0100	0000	0100	0000	0100	0000	0100
1 0	0 1	0 0	0 1	0 0	0011	0011	0011	0011	0011	0011	0011	0011

b) Tabla de transición de estados y salidas

Tabla 5.4: Asignación de estados y tabla de transición de estados y salidas del modelo de Huffman-Mealy

De forma análoga en la Tabla 5.5 se representa la tabla de estados para el modelo de Moore (deducida a partir de la Figura 5.22b). En la Tabla 5.6 se muestran la asignación de estados elegida (se requieren tres

elementos de memoria Q_2, Q_1 y Q_0) y la tabla de transición de estados y de salida que correponden a la tabla de estados de la Tabla 5.5.

Estado presente	Próximo estado							
	$\overline{I}\,\overline{s_1}\,\overline{s_0}$	$\overline{I}\,\overline{s_1}\,s_0$	$\overline{I}\,s_1\,\overline{s_0}$	$\overline{I}\,s_1\,s_0$	$I\,\overline{s_1}\,\overline{s_0}$	$I\,\overline{s_1}\,s_0$	$I\,s_1\,\overline{s_0}$	$I\,s_1\,s_0$
S_0	S_0	S_0	S_0	S_0	S_{00}	S_{00}	S_{00}	S_{00}
S_{00}	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1
S_1	S_2	S_{10}	S_2	S_{10}	S_2	S_{10}	S_2	S_{10}
S_{10}	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2
S_2	S_1	S_1	S_0	S_0	S_1	S_1	S_0	S_0

Tabla 5.5: Tabla de estados del modelo de Moore

	$Q_2\ Q_1\ Q_0$
$S_0 \rightarrow$	0 0 0
$S_{00} \rightarrow$	0 0 1
$S_1 \rightarrow$	0 1 0
$S_{10} \rightarrow$	0 1 1
$S_2 \rightarrow$	1 0 0

a) Asignación de estados

$Q_2\ Q_1\ Q_0$	Próximo estado								Salida
	$\overline{I}\,\overline{s_1}\,\overline{s_0}$	$\overline{I}\,\overline{s_1}\,s_0$	$\overline{I}\,s_1\,\overline{s_0}$	$\overline{I}\,s_1\,s_0$	$I\,\overline{s_1}\,\overline{s_0}$	$I\,\overline{s_1}\,s_0$	$I\,s_1\,\overline{s_0}$	$I\,s_1\,s_0$	$c_0\,c_1\,c_2\,c_3$
0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 1	0 0 1	0 0 1	0 0 1	0 0 0 0
0 0 1	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	0 1 0	1 0 0 1
0 1 0	1 0 0	0 1 1	1 0 0	0 1 1	1 0 0	0 1 1	1 0 0	0 1 1	0 0 0 0
0 1 1	1 0 0	1 0 0	1 0 0	1 0 0	1 0 0	1 0 0	1 0 0	1 0 0	0 1 0 0
1 0 0	0 1 0	0 1 0	0 0 0	0 0 0	0 1 0	0 1 0	0 0 0	0 0 0	0 0 1 1

b) Tabla de transición de estados y salidas

Tabla 5.6: Asignación de estados y tabla de transición de estados y salidas del modelo de Moore

En este apartado se va a utilizar, para el diseño con elementos de memoria tipo *D*, el modelo de Huffman-Mealy. El desarrollo para el modelo de Moore sigue un procedimiento paralelo. Teniendo en cuenta la función de excitación de un elemento de memoria tipo *D*:

$$D(t) = Q(t+1)$$

y la tabla de transición de estados de la Tabla 5.4b, se deducen las expresiones de las entradas D_1 y D_0 a los dos elementos de memoria (ver Figura 5.25).

Las señales de control c_i (i = 0, 1, 2 y 3) de la unidad de control del multiplicador se deducen directamente de las expresiones lógicas (5.3). En la Figura 5.26 se muestra la realización del circuito secuencial de la unidad de control del multiplicador para la asignación de estados realizadas.

$$\begin{aligned} c_0 &= S_0\,I = \overline{Q}_1\overline{Q}_0\,I \\ c_1 &= S_1\,s_0 = \overline{Q}_1 Q_0\,s_0 \\ c_2 &= S_2 = Q_1\overline{Q}_0 \\ c_3 &= c_0 + c_2 \end{aligned} \tag{5.4}$$

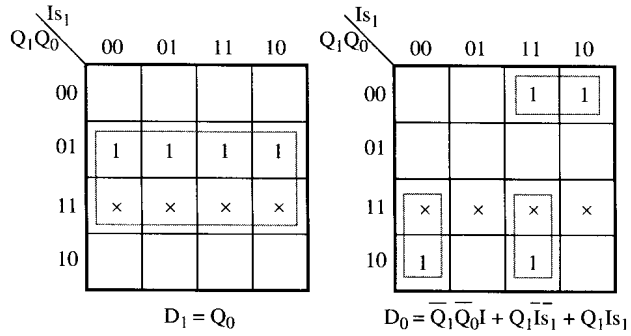


Figura 5.25: Funciones de entrada a los elementos de memoria tipo D

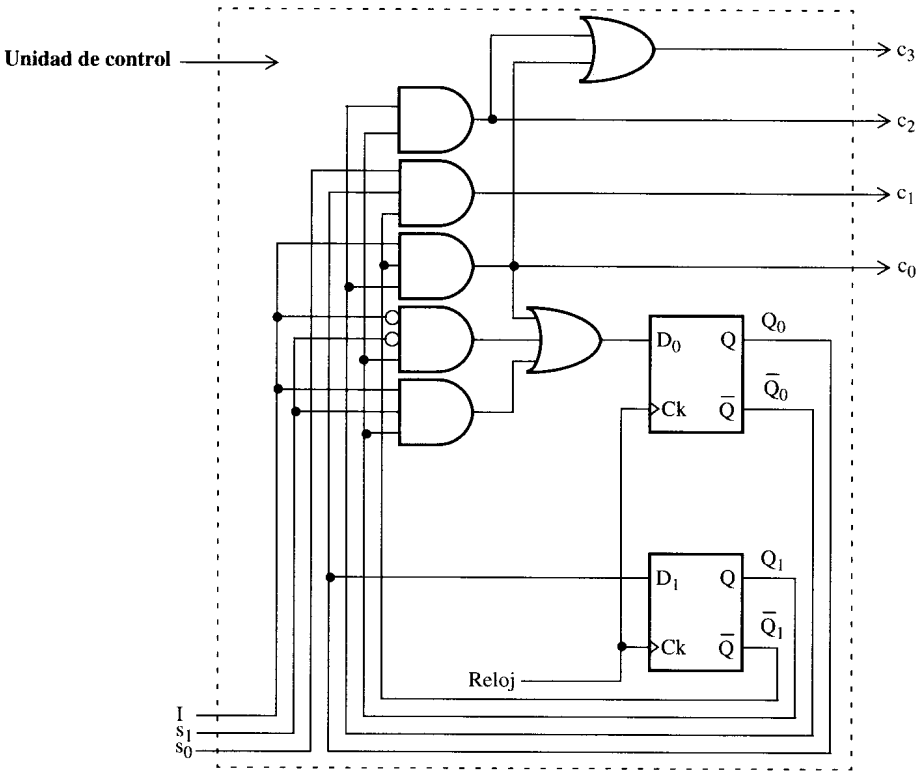


Figura 5.26: Circuito secuencial de la unidad de control del multiplicador

5.5.6 Unidad de control diseñada con un registro de secuencia y un decodificador

En este apartado se va a sintetizar la unidad de control mediante el modelo de Huffman-Mealy utilizando un registro de secuencia y un decodificador para proporcionar una señal de salida para cada uno de los estados. Un registro con n elementos de memoria puede tener hasta 2^n estados y un decodificador de n entradas tiene

2^n salidas, que permite asociar una a cada uno de los estados. También para generar las señales de control se puede necesitar cierta lógica combinacional adicional. El método, aunque se explica para el caso del multiplicador binario es completamente general. La parte de secuenciamiento del diagrama ASM del multiplicador binario tiene tres estados (S_0, S_1 y S_2) y dos entradas (I y s_1). Para implementar el diagrama ASM con un registro de secuencia y un decodificador se necesitan dos elementos de memoria para el registro y un decodificador de *2* a *4* líneas. Como hay tres estados, sólo se emplean tres de las cuatro salidas del decodificador.

En la Tabla 5.7 se muestra la tabla de estados de la parte de secuenciamiento de la unidad de control que se deduce directamente del diagrama ASM modificado de la Figura 5.23. Se representan los elementos de memoria por Q_1 y Q_0 y se codifica cada estado S_i igual que en el apartado anterior mediante el número binario i, de forma que $S_0 = 00$, $S_1 = 01$ y $S_2 = 10$. Se observa que las columnas de las entradas tienen valores no especificados (×) siempre que la variable de entrada no se utiliza para determinar el próximo estado. Las salidas de la parte de secuenciamiento de la unidad de control (que son las salidas del decodificador) se identifican con los nombres de los estados S_0, S_1 y S_2. El código binario para el estado presente determina la salida del decodificador que es igual a 1 en un instante de tiempo dado. Así cuando el estado presente es $Q_1Q_0 = 00$, la salida número 0 del decodificador (que identifica al estado S_0) es 1 mientras que las otras salidas del decodificador son igual a 0.

Estado presente			Entradas		Próximo estado		Salidas decodificador		
Nombre	Q_1	Q_0	I	s_1	Q_1	Q_0	S_0	S_1	S_2
S_0	0	0	0	×	0	0	1	0	0
	0	0	1	×	0	1	1	0	0
S_1	0	1	×	×	1	0	0	1	0
S_2	1	0	×	0	0	1	0	0	1
	1	0	×	1	0	0	0	0	1
-	1	1	×	×	×	×	×	×	×

Tabla 5.7: Tabla de estado de la parte de secuenciamiento de la unidad de control del multiplicador

Este ejemplo tiene un número pequeño de estados y de entradas, sin embargo en la mayoría de las aplicaciones de diseño de unidades de control el número de estados es mucho mayor. La aplicación del método convencional que se ha expuesto en el apartado anterior requeriría entonces un trabajo excesivo para obtener las expresiones simplificadas de las entradas a los elementos de memoria. El diseño se puede facilitar si se toma en consideración el hecho de que se tienen disponibles las salidas del decodificador. Estas salidas proporcionan una única señal para representar cada uno de los posibles estados presentes del circuito. Además en lugar de emplear los diagramas de Karnaugh para simplificar las ecuaciones de los elementos de memoria se pueden obtener estas directamente por simple inspección de la tabla de estado. Por ejemplo, de las condiciones para el próximo estado dadas en la Tabla 5.7 se observa que el próximo estado de Q_0 es igual a 1 cuando el estado presente es S_0 y la entrada I es igual a 1 o bien cuando el estado presente es S_2 y la entrada s_1 es igual a 0. Estas condiciones se pueden expresar mediante la ecuación lógica siguiente:

$$D_0 = S_0 I + S_2 \bar{s}_1$$

para la entrada D del elemento de memoria Q_0. Análogamente la entrada D del elemento de memoria Q_1 es:

$$D_1 = S_1$$

Conviene observar que estas ecuaciones derivadas por inspección de la tabla de estado utilizan los propios estados (S_0, S_1 y S_2) en lugar de las variables de estado (Q_1 y Q_0) ya que en este método se dispone del decodificador que genera los estados. En algunos casos es posible encontrar ecuaciones más simples para las entradas a los elementos de memoria tipo D utilizando directamente las variables de estado (Q_i) en lugar de los estados (S_i). En la Figura 5.27 se muestra el diagrama lógico de la unidad de control del multiplicador deducida por este procedimiento.

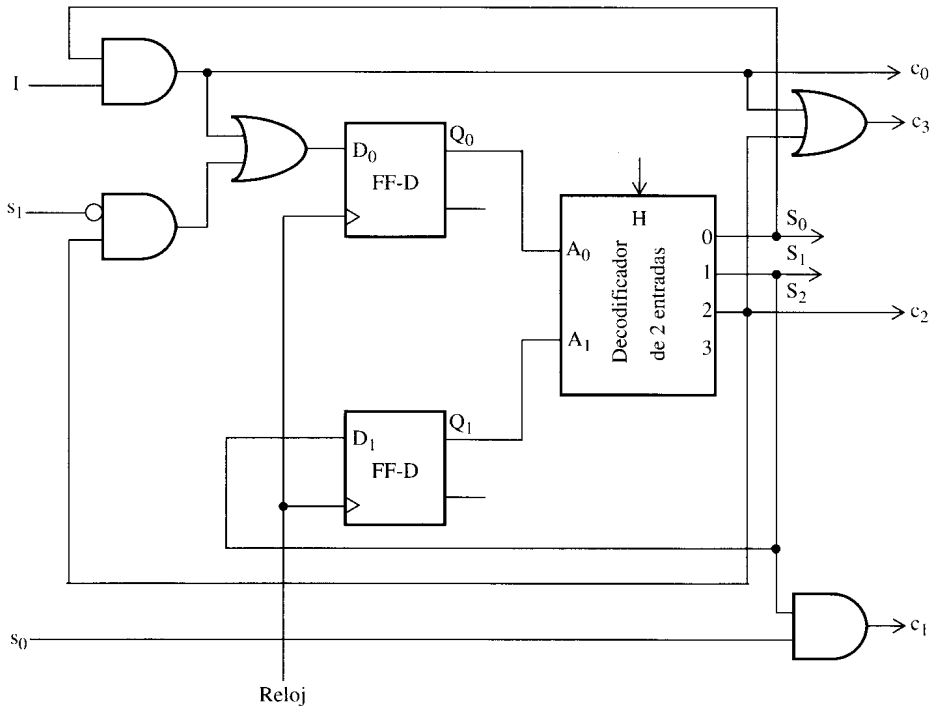


Figura 5.27: Diagrama lógico de la unidad de control del multiplicador

Consta de un registro de dos bits con los elementos de memoria Q_1 y Q_0 y un decodificador de 2 a 4. Las tres salidas del decodificador se utilizan para generar las señales de control así como las entradas a los elementos de memoria Q_1 y Q_0 que determinan el próximo estado. Las salidas *Inicializar* (c_0), *Borrar_C* (c_3), *Desplazar_Dec* (c_2) y *Sumar_Cargar* (c_1) se determinan a partir de la Tabla 5.2. Las señales de control c_0 y c_2 están ya disponibles, así que se añaden las salidas convenientemente etiquetadas. Sin embargo, como se muestra en la Figura 5.27, se deben añadir ciertas puertas lógicas para obtener las señales de control c_1 y c_3. El diseño del multiplicador binario se completa conectando las salidas de la unidad de control con las entradas de control en la unidad de procesamiento (ver Figura 5.28).

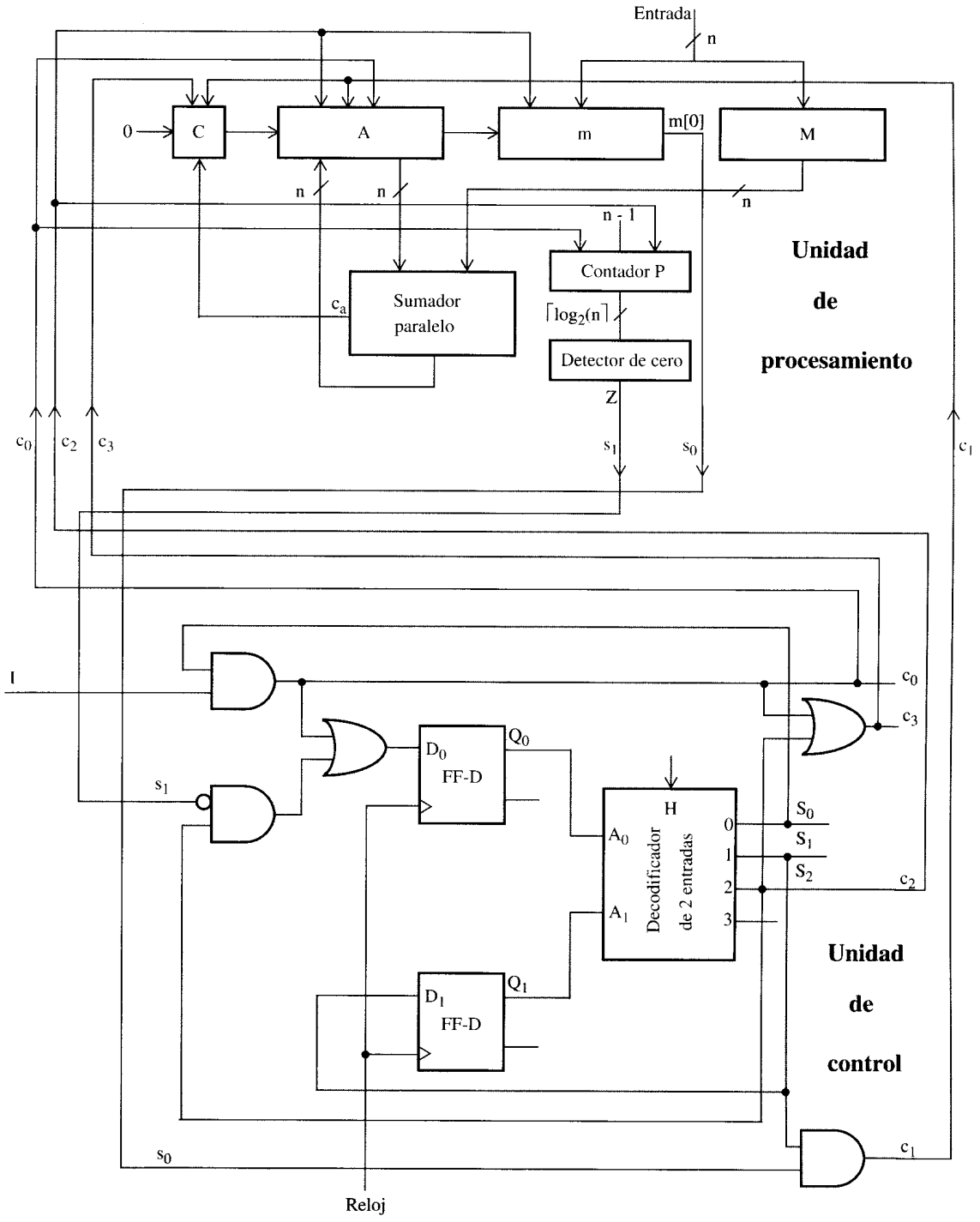


Figura 5.28: Diagrama completo del multiplicador binario

5.5.7 Unidad de control diseñada con un elemento de memoria por estado

Otro método para diseñar la unidad de control consiste en asignar un elemento de memoria a cada uno de los estados de forma que en cualquier instante de tiempo sólo uno de ellos tiene como salida el valor 1 y el resto valen 0. Cuando el 1 está en el elemento de memoria que corresponde a un estado particular, el circuito secuencial está en ese mismo estado. El único 1 que existe en los elementos de memoria se propaga de un elemento de memoria a otro bajo el control de la lógica de decisión que proporciona el diagrama ASM. Con esta configuración cada elemento de memoria representa un estado que es el estado presente cuando su salida es 1. Resulta obvio que este método utiliza para el diseño del circuito secuencial el número máximo de elementos de memoria. Por ejemplo, un circuito secuencial con 16 estados que utiliza una codificación de variables mínima sólo requiere 4 elementos de memoria. Con un elemento de memoria por estado, el circuito precisa de 16 elementos de memoria, uno para cada estado. A primera vista puede parecer que este método aumentaría el coste del sistema ya que se utilizan más elementos de memoria. Sin embargo el método ofrece algunas ventajas. Una de ella es la simplicidad con la que se diseña el circuito, simplemente inspeccionando el diagrama ASM. Si se emplean elementos de memoria tipo *D* no se necesitan utilizar las tablas de transición de estados ni calcular las ecuaciones de entrada a los elementos de memoria.

En la Figura 5.29 se muestran las reglas de sustitución para transformar un diagrama ASM en un circuito secuencial con un elemento de memoria por estado. Estas reglas se aplican más fácilmente sobre el diagrama ASM modificado que representa sólo la información del secuenciamiento de estados. Cada regla especifica la sustitución de un bloque de un diagrama ASM por un circuito lógico. Tal como se observa en la Figura 5.29a, el bloque de estado se reemplaza por un elemento de memoria etiquetado con el nombre del estado. La entrada (salida) al bloque de estado se corresponde con la entrada (salida) al elemento de memoria.

En la Figura 5.29b, el bloque de decisión se sustituye por un demultiplexor. La señal que corresponde a la entrada del bloque de decisión se envía a una de las dos salidas del circuito dependiendo del valor de la señal *V*. Si *V* es 0, la señal se envía a la *Salida 0* y si *V* es 1 a la *Salida 1*. El demultiplexor actúa como un conmutador que dirige el valor que tiene a su entrada de acuerdo con los diferentes caminos que se establecen en el diagrama ASM.

La unión en la Figura 5.29c es cualquier punto en el cual se unen dos o más líneas dirigidas de un diagrama ASM. Si un 1 está presente en el circuito en cualquier línea que se corresponde a uno de los caminos de entrada, entonces también debe aparecer sobre la línea que corresponde al camino de salida, lo que da a esa línea el valor 1. Si ninguna de las líneas que corresponden a un camino de entrada en la unión tienen el valor 1, entonces la línea de salida debe tener el valor 0. De esta forma la unión se reemplaza por una puerta OR.

Con estas tres transformaciones, la parte de secuenciamiento del diagrama ASM se puede sustituir por simple inspección por un circuito con un elemento de memoria por estado. La parte de salida del circuito secuencial se obtiene directamente del diagrama ASM original o de la tabla de las señales de control deducidas del propio diagrama. En la Figura 5.29d se ilustra la salida de un bloque de condición que se sustituye simplemente por una conexión en el circuito.

A continuación se utilizan estas transformaciones para determinar la unidad de control, con un elemento de memoria por estado, para el multiplicador binario. Se emplea el diagrama ASM modificado de la Figura 5.23 para la parte de secuenciamiento. Obsérvese que en este caso se ignora la codificación binaria de los estados que se realizó en el diseño anterior. En la Figura 5.30 se muestra el circuito lógico resultante.

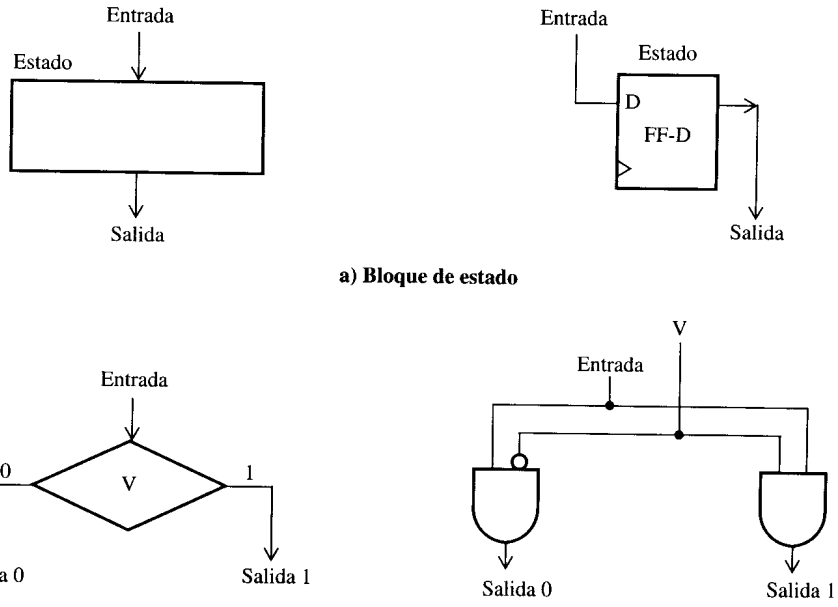


Figura 5.29: Reglas de transformación para una unidad de control con un elemento de memoria por estado

Los pasos que se han seguido son los siguientes:

1) En primer lugar se sustituyen los tres bloques de estado por tres elementos de memoria tipo *D* que se etiquetan con el nombre del estado. En la Figura 5.30 se representa esta acción por un 1 encerrado en un círculo.

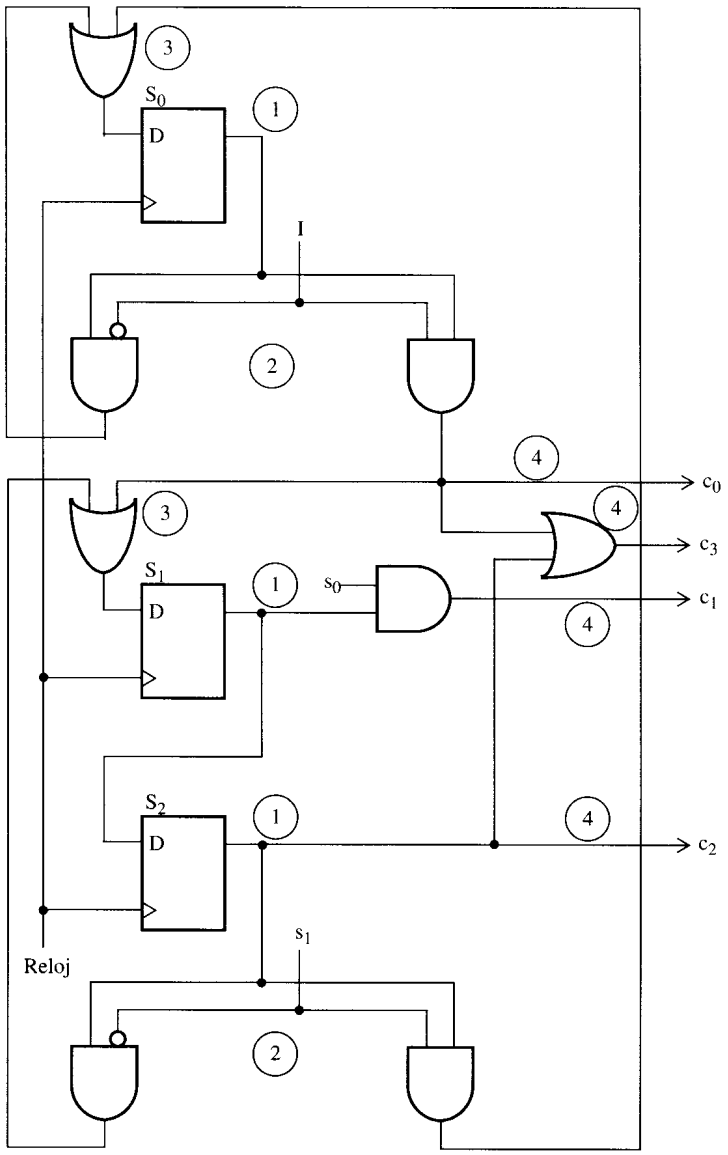


Figura 5.30: Unidad de control con un elemento de memoria por estado para el multiplicador binario

2) Los bloques de decisión se reemplazan por un demultiplexor con la variable de decisión como su entrada de selección. Esta acción se representa en la Figura 5.30 por un 2 encerrado en un círculo.

3) Los puntos de unión se transforman en una puerta OR. Esta acción se representa en la Figura 5.30 por un 3 encerrado en un círculo.

4) Finalmente se forman las señales de control, que son las salidas del circuito, de acuerdo con las ecuaciones lógicas obtenidas en la Tabla 5.2. De esta tabla se deduce que la función booleana de la salida *Inicializar* (c_0) está ya disponible en el diagrama lógico, así que se añade al diagrama del circuito. De igual forma se puede añadir la salida *Desplazar_Dec* (c_2). Para las salidas *Borrar_C* (c_3) y *Sumar_Cargar* (c_1) se necesitan añadir ciertas puertas lógicas para implementar sus funciones booleanas. Esta acción se representa en la Figura 5.30 por un 4 encerrado en un círculo.

Una cuestión final en el diseño de la unidad de control con un elemento de memoria por estado es la fase de inicialización del estado que tiene que colocar un 1 en el elemento de memoria S_0 y un 0 en todos los otros. Esto se puede hacer utilizando una entrada asíncrona de PRESET (puesta a 1) en el elemento de memoria S_0 y una entrada asíncrona de CLEAR (puesta a 0) en los otros elementos de memoria. Si los elementos de memoria sólo disponen de una entrada asíncrona de CLEAR en lugar de las dos, se coloca una puerta inversora (NOT) justamente antes de la entrada D y otra a la salida del elemento de memoria S_0. De esta forma el elemento de memoria S_0 contendrá realmente un 0 cuando se está en el estado S_0 y un 1 en los restantes estados. Esto permite utilizar la entrada CLEAR para inicializar los tres elementos de memoria en el circuito. Conviene tener presente que la utilización de las entradas asíncronas de los elementos de memoria para otra función distinta de la de inicialización del circuito es generalmente una mala técnica de diseño.

Una vez completado el diseño de la lógica de control con un elemento de memoria por estado, se puede optimizar. Por ejemplo, si hay una serie de uniones conectadas juntas por líneas, se combinan las puertas OR que resultan de la transformación. También los demultiplexores en cascada se pueden combinar en uno sólo.

5.5.8 Unidad de control diseñada utilizando un registro de estado y una memoria ROM

La memoria ROM proporciona una realización de la unidad de control más flexible y directa que las consideradas anteriormente. Las distintas arquitecturas que incorporan una memoria ROM difieren en la forma en que tratan a las variables de condición.

En la Figura 5.31 se muestra el esquema básico de un circuito secuencial con una memoria ROM que sintetiza una unidad de control. Todas las variables de condición y señales externas se utilizan como parte de las líneas de dirección de la memoria ROM. Si en total hay q de estas variables entonces cada uno de los estados requerirá 2^q palabras en la memoria ROM. Para n variables de estado hay 2^n grupos de 2^q palabras, lo que hace un total de 2^{n+q} palabras. Cada palabra de la memoria ROM consta de dos campos, uno de n bits que representa el próximo estado y otro de m bits que constituye las señales de control.

Con la estructura de la Figura 5.31 la realización de la unidad de control del multiplicador binario cuando se modela como un circuito secuencial tipo Huffman-Mealy necesita una memoria ROM de 2^{2+3} palabras de 2 + 4 bits (32 × 6 bits). En este caso $n = 2$ (la unidad de control posee tres estados y se necesitan 2 bits para su codificación), $q = 3$ (las entradas a la unidad de control son I, s_0 y s_1) y $m = 4$ (las salidas de la unidad de control son c_0, c_1, c_2 y c_3). En la Tabla 5.8 se muestra el contenido de la memoria ROM. La asignación del estado S_i corresponde a la representación binaria del número entero i ($i = 0, 1, 2$). Se observa que sólo es necesario rellenar las 24 primeras posiciones de la memoria ROM debido a que el estado que se codifica como 1 1, que correspondería a S_3, no se necesita en este caso.

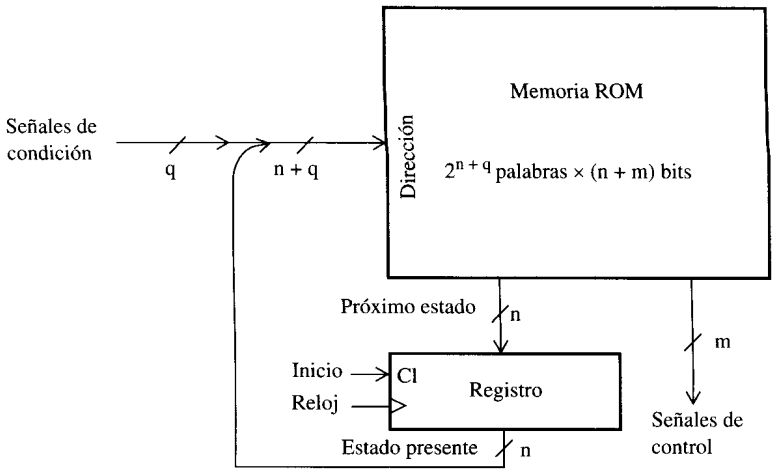


Figura 5.31: Estructura básica de un circuito secuencial con una memoria ROM

Este método de síntesis de la unidad de control resulta bastante ineficaz desde el punto de vista de utilización de la memoria ROM, ya que sólo unas pocas condiciones son relevantes en cada estado y en consecuencia muchas palabras de la memoria poseen el mismo contenido. La ventaja principal de este esquema es que la transformación del diagrama de transición de estados de la Figura 5.4b, en el contenido de la memoria ROM correspondiente, es directa. Permite realizar en un sólo paso cualquier ruptura de secuencia de las 2^q que son posibles con q variables de condición o señales externas.

También se podría haber utilizado un diagrama de estado tipo Moore en cuyo caso las señales de control que corresponden a un mismo estado presente serían idénticas en la memoria ROM. En la implementación de la memoria ROM que se muestra en la Tabla 5.8 (que se modela como un circuito secuencial tipo Huffman-Mealy) las señales de control especificadas en cada palabra son diferentes según lo requieran las variables de condición. Así por ejemplo el contenido de las posiciones de memoria 3 y 4 que corresponden a un mismo estado presente (S_0 = 00) dan como salidas señales de control distintas 0000 y 1001 respectivamente.

Selección por estado

Un método para reducir el tamaño de la memoria ROM se basa en el hecho de que para cada estado no se necesitan todas las variables de condición. Si sólo se precisa una variable de condición por estado (cosa que sucede en el ejemplo del multiplicador) es mejor seleccionar dicha variable fuera de la memoria ROM. Esto reduce el tamaño de la memoria ROM a 2 palabras por estado, lo que da un tamaño total de 2^{n+1} palabras.

Para seleccionar la variable de condición que se precisa en cada estado se puede emplear un multiplexor controlado por las variables de estado. Si en un estado no se requiere ninguna variable de condición la correspondiente entrada del multiplexor se puede conectar a 0 ó 1, de forma que el próximo estado estará respectivamente en la dirección par o impar de la memoria ROM. En la Figura 5.32 se muestra esta variante de la estructura básica de la Figura 5.31 que sólo necesita una memoria ROM de 2^{n+1} palabras de $n + m$ bits.

Dirección de la ROM			Contenido de la ROM	
	Estado presente	I s_1 s_0	Próximo estado	c_0 c_1 c_2 c_3
0	0 0	0 0 0	0 0	0 0 0 0
1	0 0	0 0 1	0 0	0 0 0 0
2	0 0	0 1 0	0 0	0 0 0 0
3	0 0	0 1 1	0 0	0 0 0 0
4	0 0	1 0 0	0 1	1 0 0 1
5	0 0	1 0 1	0 1	1 0 0 1
6	0 0	1 1 0	0 1	1 0 0 1
7	0 0	1 1 1	0 1	1 0 0 1
8	0 1	0 0 0	1 0	0 0 0 0
9	0 1	0 0 1	1 0	0 1 0 0
10	0 1	0 1 0	1 0	0 0 0 0
11	0 1	0 1 1	1 0	0 1 0 0
12	0 1	1 0 0	1 0	0 0 0 0
13	0 1	1 0 1	1 0	0 1 0 0
14	0 1	1 1 0	1 0	0 0 0 0
15	0 1	1 1 1	1 0	0 1 0 0
16	1 0	0 0 0	0 1	0 0 1 1
17	1 0	0 0 1	0 1	0 0 1 1
18	1 0	0 1 0	0 0	0 0 1 1
19	1 0	0 1 1	0 0	0 0 1 1
20	1 0	1 0 0	0 1	0 0 1 1
21	1 0	1 0 1	0 1	0 0 1 1
22	1 0	1 1 0	0 0	0 0 1 1
23	1 0	1 1 1	0 0	0 0 1 1

Tabla 5.8: Contenido de la memoria ROM de la unidad de control del multiplicador

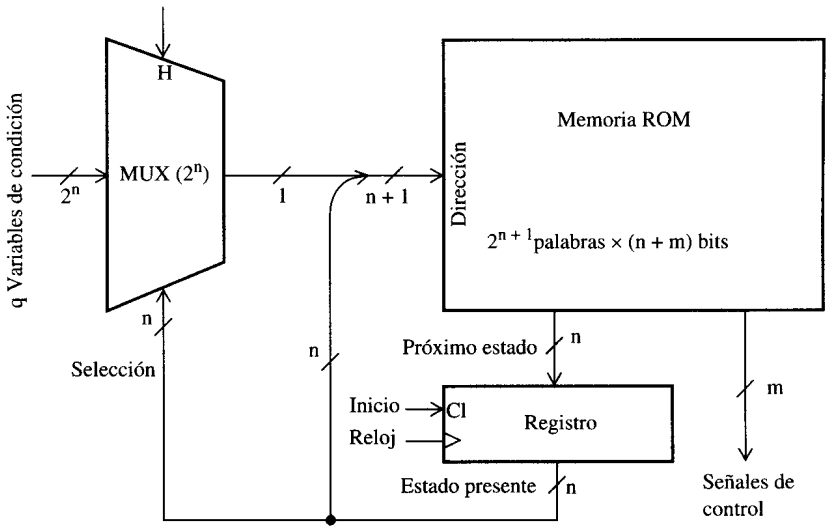


Figura 5.32: Unidad de control con una memoria ROM y un multiplexor (selección por estado)

En la Figura 5.33 se muestra el caso de la unidad de control del multiplicador binario, que necesita una memoria ROM de $2^{2+1} = 8$ palabras (de las que sólo se necesitan 6 palabras) de 2 + 4 = 6 bits, un multiplexor de 4 entradas y un registro de 2 bits que almacena el próximo estado. La asignación del estado S_i corresponde a la representación binaria del número entero i. En la Tabla 5.9 se muestra el contenido de la memoria ROM.

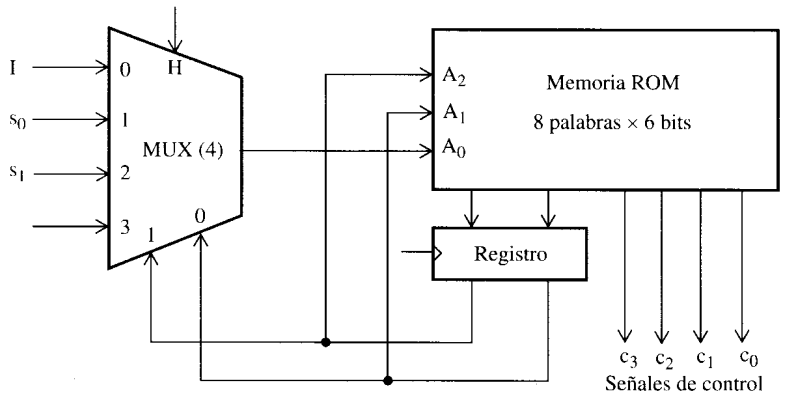


Figura 5.33: Unidad de control del multiplicador (selección por estado)

Dirección de la ROM			Contenido de la ROM	
	Estado presente	Señal de condición	Próximo estado	$c_0\ c_1\ c_2\ c_3$
0	0 0	0 (I = 0)	0 0	0 0 0 0
1	0 0	1 (I = 1)	0 1	1 0 0 1
2	0 1	0 (s_0 = 0)	1 0	0 0 0 0
3	0 1	1 (s_0 = 1)	1 0	0 1 0 0
4	1 0	0 (s_1 = 0)	0 1	0 0 1 1
5	1 0	1 (s_1 = 1)	0 0	0 0 1 1

Tabla 5.9: Contenido de la memoria ROM (método de selección por estado)

Selección por campo

Una alternativa a este último diseño, que se conoce como selección por campo, es utilizar un multiplexor de q entradas y $log_2 q$ variables de control que permiten seleccionar una condición para cada estado. El tamaño de la memoria ROM que se necesita es de 2^{n+1} palabras con una longitud de $log_2 q + n + m$ bits, ya que hay que incluir en la palabra el código de la variable de control del multiplexor que se asocia a la variable de condición seleccionada. En la Figura 5.34 se muestra una realización genérica de este tipo. El método de selección por campo es preferible al de selección por estado si el número total de variables de condición es más pequeño que el número de estados (el multiplexor posee menos entradas).

En la Figura 5.35 se muestra la unidad de control del multiplicador realizada con este último método. Se necesita una ROM de $2^{2+1} = 8$ palabras de 2+ 2 + 4 = 8 bits, un multiplexor de 4 entradas y un registro de 4 bits que almacena, además del próximo estado, el número de la variable de condición que hay que selec-

cionar. La asignación del estado S_i corresponde a la representación binaria del número entero i. La entrada asíncrona del registro *Clear* inicializa a la unidad de control en el estado S_0 con la orden externa de *Inicio*.

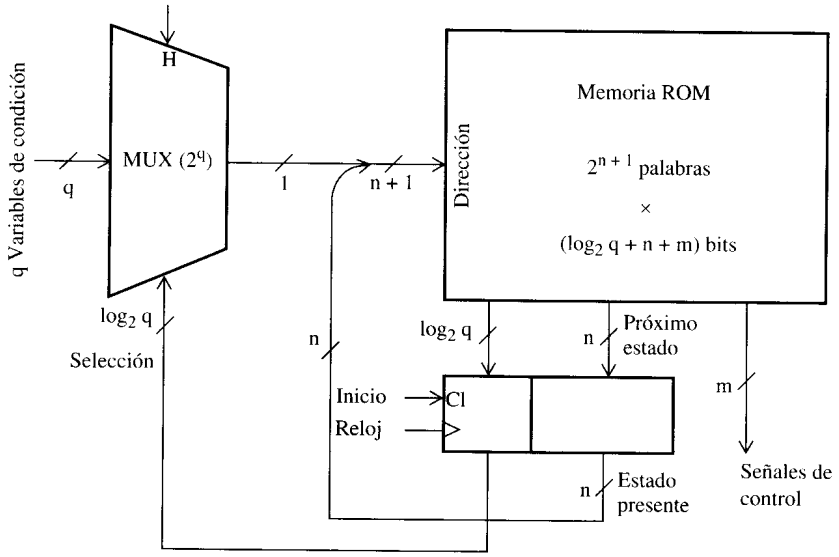


Figura 5.34: Unidad de control con una memoria ROM y un multiplexor (método de selección por campo)

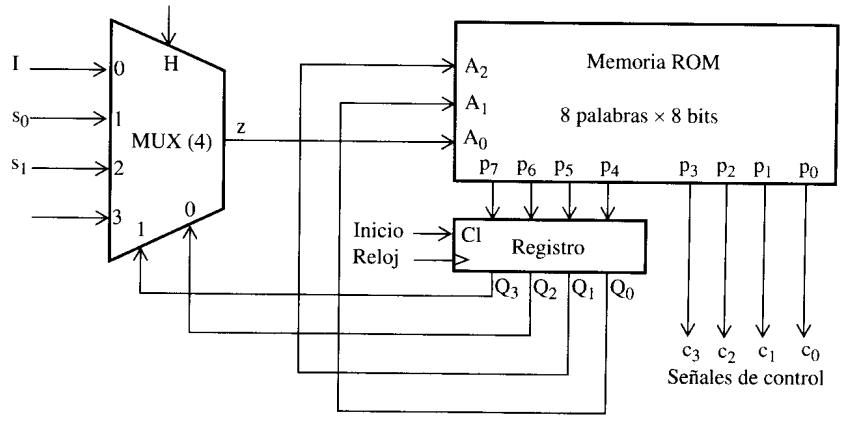


Figura 5.35: Unidad de control del multiplicador (selección por campo)

En el método de selección por campo la dirección del estado presente A se define como:

$$A = (A_2, A_1, A_0) = (Q_1, Q_0, z)$$

donde z corresponde a la condición seleccionada. El contenido de una palabra p de la memoria ROM es:

$$p = (p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0)$$

en esta palabra p_7 y p_6 definen la condición que hay que seleccionar en el próximo estado, (p_5, p_4) son los dos bits más significativos de la dirección de la memoria ROM y (p_3, p_2, p_1, p_0) son las señales de control. En la Tabla 5.6 se muestra el contenido de la memoria ROM . Si se compara el contenido de la memoria ROM de la Tabla 5.9 (selección por estado) con el de la Tabla 5.10 (selección por campo) se observa que este último método no supone, en este caso, ninguna ventaja con respecto al de selección por estado. De hecho los bits p_7 y p_6 de cada palabra de la memoria ROM que permiten seleccionar la entrada de datos del multiplexor que se conecta a su salida z son idénticos a los bits p_5 y p_4 que definen el próximo estado.

Dirección de la ROM			**Contenido de la ROM**	
	Estado presente	Señal de condición	p_7 p_6 p_5 p_4	p_3 p_2 p_1 p_0 c_0 c_1 c_2 c_3
0	0 0	0 (I = 0)	0 0 0 0	0 0 0 0
1	0 0	1 (I = 1)	0 1 0 1	1 0 0 1
2	0 1	0 (s_0 = 0)	1 0 1 0	0 0 0 0
3	0 1	1 (s_0 = 1)	1 0 1 0	0 1 0 0
4	1 0	0 (s_1 = 0)	0 1 0 1	0 0 1 1
5	1 0	1 (s_1 = 1)	0 0 0 0	0 0 1 1

Tabla 5.10: Contenido de la memoria ROM (método de selección por campo)

5.5.9 Unidad de control diseñada utilizando un contador y un decodificador

En determinadas situaciones la utilización de un contador (ver Figura 5.36) para almacenar el estado del sistema, en lugar de emplear un simple registro, puede conducir a una simplificación significativa de la lógica combinacional que se necesita para la generación del próximo estado. Esta circunstancia se presenta cuando la secuencia de estados que sigue la máquina es bastante parecida a la secuencia de los números naturales, 0, 1, 2, ... ya que en ese caso para realizar la transición del estado n al estado n+1, solamente habrá que activar la señal de cuenta del contador. La utilización de un contador como registro de estado en la síntesis de un circuito secuencial se basa en las siguientes consideraciones:

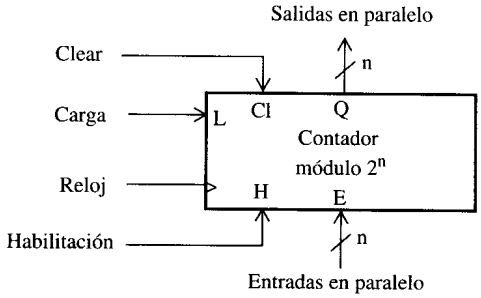
a) Si las transiciones entre estados presentan una estructura lineal, se pueden realizar fácilmente programando el contador para que esté en el modo de "contaje" (es decir, que incremente su contenido en una unidad con cada pulso de reloj). Esto se hace poniendo a 1 la señal de habilitación H del contador.

b) Las transiciones entre estados que suponen una ruptura de la estructura lineal se efectúan programando el contador para que esté en el modo de "carga paralela" (L = 1).

Se puede aplicar esta idea al diseño de la unidad de control del multiplicador que en este caso se va a modelar como un circuito secuencial tipo Moore. El estado se almacenará en un contador de tres bits que responde al esquema de la Figura 5.36. Las señales de habilitación (H), carga (L) y las entradas en paralelo al contador (E_2, E_1 y E_0) se pueden obtener de forma directa del diagrama de transición de estados de la Figura 5.6b.

En la Tabla 5.7 se muestra la determinación de las entradas del contador y de las señales H y L. Tal como refleja la Tabla 5.7, de forma genérica la transición del estado S_i al estado S_j se hace activando la señal de habilitación H cuando la codificación de ambos estados corresponde a números enteros consecutivos (por ejemplo el estado presente es S_{00} = 001, el próximo estado será S_1 = 010). Para este tipo de transiciones el

valor que tenga la entrada paralelo al contador es irrelevante (se utiliza el símbolo × siempre que dicho valor no se utiliza para la determinación del próximo estado). Evidentemente, esto produce un elevado porcentaje de minterms redundantes que pueden aprovecharse si se desea para simplificar la lógica combinacional. Cuando en la transición del estado S_i al estado S_j la codificación de ambos estados no corresponde a números enteros consecutivos (por ejemplo cuando $s_0 = 0$ y el estado presente es $S_1 = 010$, el próximo estado será $S_2 = 100$) hay que cargar las entradas paralelas del contador con la codificación asignada al próximo estado S_j (en el ejemplo anterior $E_2E_1E_0 = 100$).



$$\mathbf{Q}(t+1) = \begin{cases} 0 & \text{si Clear} = 1 \text{ (asíncrona)} \\ \mathbf{E}(t) & \text{si L} = 1 \\ (\mathbf{Q}(t) + 1) \bmod 2^n & \text{si H} = 1 \text{ y L} = 0 \\ \mathbf{Q}(t) & \text{en cualquier otro caso} \end{cases}$$

Figura 5.36: Diagrama de bloques de un contador módulo 2^n

Estado presente	**Condición**	**Próximo estado**	**H**	**L**	**Entradas en paralelo** $E_2\ E_1\ E_0$	**Señales de control** $c_0\ c_1\ c_2\ c_3$
S_0 (000) S_0 (000)	I = 0 I = 1	S_0 (000) S_{00} (001)	0 1	× 0	× × × × × ×	0 0 0 0 0 0 0 0
S_{00} (001)	×	S_1 (010)	1	0	× × ×	1 0 0 1
S_1 (010) S_1 (010)	$s_0 = 0$ $s_0 = 1$	S_2 (100) S_{10} (011)	× 1	1 0	1 0 0 × × ×	0 0 0 0 0 0 0 0
S_{10} (011)	×	S_2 (100)	1	0	× × ×	0 1 0 0
S_2 (100) S_2 (100)	$s_1 = 0$ $s_1 = 1$	S_1 (010) S_0 (000)	× ×	1 1	0 1 0 0 0 0	0 0 1 1 0 0 1 1

Tabla 5.11: Determinación de las entradas de control del contador

En este caso resulta muy simple obtener la realización del circuito secuencial de la unidad de control que se muestra en la Figura 5.37. Consta de un contador módulo 8, un decodificador binario de 3 entradas y unas cuantas puertas lógicas. La entrada asíncrona del contador *Clear* (poner a 0) inicializa la unidad de control en el estado S_0. Las salidas de la parte de secuenciamiento de la unidad de control (que son las salidas del

decodificador) se identifican con los nombres de los estados S_0, S_{00}, S_1, S_{10} y S_2. La asignación de estados efectuada (ver Figura 5.6a) determina la salida del decodificador que es igual a 1 en un instante de tiempo dado. Así cuando el estado presente es $Q_2Q_1Q_0 = 011$, la salida número 3 del decodificador (que identifica al estado S_{10}) es 1 mientras que las otras salidas del decodificador son igual a 0.

Como en el caso de la unidad de control con un registro de secuencia y un decodificador (ver apartado 5.5.6) el diseño se puede facilitar si se utilizan directamente las salidas del decodificador. Estas salidas representan cada uno de los posibles estados presentes del circuito. En la Tabla 5.11 se observa que las señales de control del contador L (carga) y H (habilitación) son complementarias. Las funciones lógicas de L, H y las entradas en paralelo al contador E_2, E_1 y E_0 se pueden obtener por simple inspección de la Tabla 5.11. Así la señal L de carga del contador es igual a 1 cuando el estado presente es S_1 y la señal de condición s_0 es igual a 0 o bien cuando el estado presente es S_2. La entrada E_2 al contador vale 1 cuando el estado presente es S_1 y la señal de condición s_0 es igual a 0. La entrada E_1 al contador vale 1 cuando el estado presente es S_2 y la señal de condición s_1 es igual a 0. La entrada E_0 al contador siempre vale 0. Estas condiciones se pueden expresar mediante las ecuaciones lógicas siguientes:

$$L = \bar{s}_0 S_1 + S_2$$

$$H = \overline{L}$$

$$E_2 = \bar{s}_0 S_1$$

$$E_1 = \bar{s}_1 S_2$$

$$E_0 = 0$$

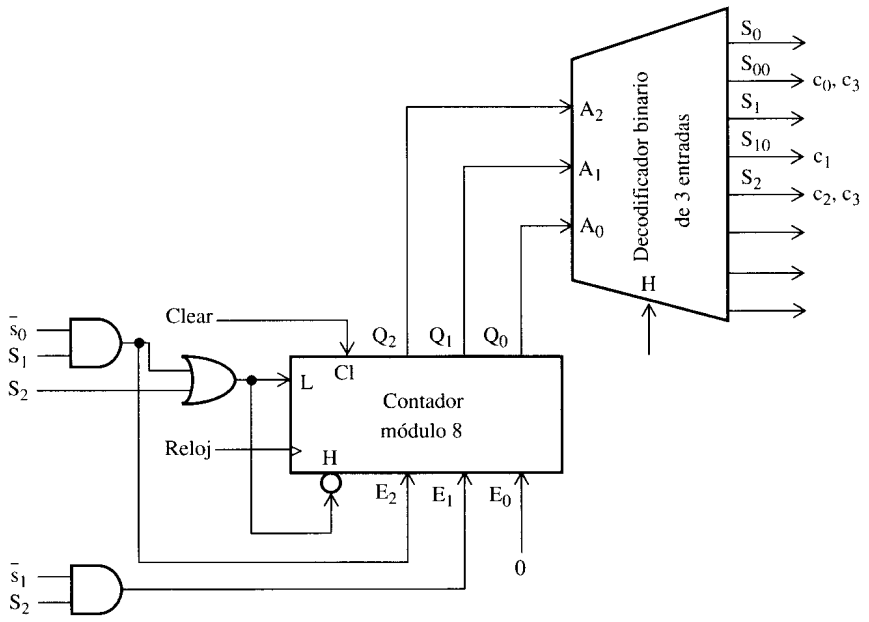


Figura 5.37: Realización de la unidad de control del multiplicador con un contador y un decodificador

Conviene insistir otra vez que estas ecuaciones lógicas deducidas directamente de la Tabla 5.11 emplean los propios estados (S_0, S_{00}, S_1, S_{10} y S_2) en lugar de las variables de estado (Q_2, Q_1 y Q_0) ya que está disponible el decodificador que genera los estados. En algunos casos pueden obtenerse ecuaciones lógicas más sencillas si se usan las variables de estado (Q_i) en lugar de los estados (S_i). En la Figura 5.37 se muestra el diagrama lógico de la unidad de control del multiplicador deducida por este procedimiento.

5.5.10 Unidad de control diseñada utilizando un registro de estado y un PLA

En la realización de la unidad de control basada en memorias ROM's, se ha considerado que cada una de las señales de control se representaba por un bit diferente de la palabra de la memoria ROM. Este planteamiento resulta ineficaz si en cada estado sólo están activas pocas señales de control (lo que sucedía en el ejemplo del multiplicador).

Una posibilidad de resolver el problema es almacenar las señales de control de forma codificada en la memoria ROM y utilizar decodificadores externos para generar las señales de control individuales. Esta estrategia se considerará en el tema 7 dedicado a la *microprogramación*.

Otra forma de solucionar el inconveniente es utilizar un array lógico programable o PLA (ver sección A-6 del apéndice A). Los PLA´s son módulos que se adaptan muy bien al diseño de unidades de control. Las alternativas en la utilización de un array lógico programable son análogas al caso de las memorias ROM's, por lo que sólo se considerará la estructura básica que se muestra en la Figura 5.38.

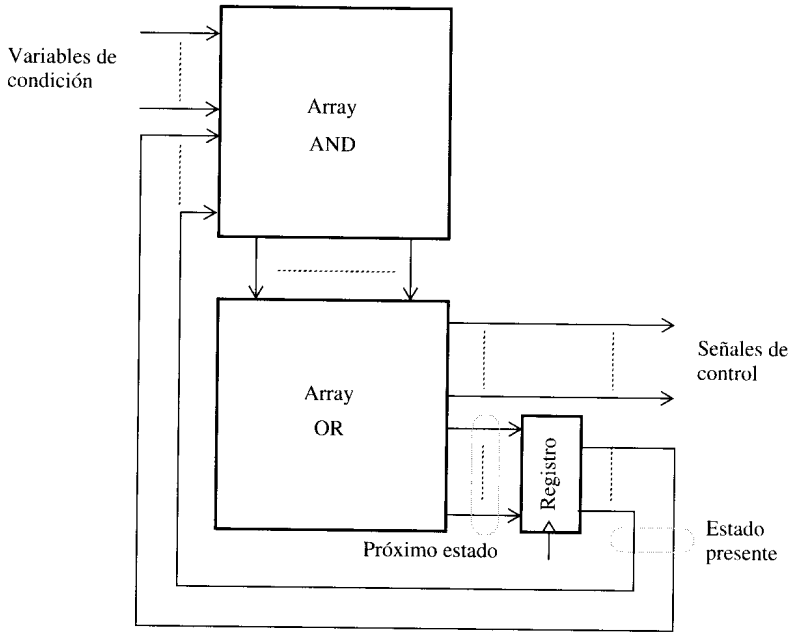


Figura 5.38: Realización de una unidad de control con un PLA y un registro de estado

Las variables de condición y el estado presente son las entradas al Array AND del PLA. El próximo estado y las señales de control son las salidas del Array OR. La transformación de un diagrama de estado en

una realización con un PLA y un registro es directa y consta de los pasos siguientes:

1) En el array OR debe existir una puerta OR de salida por cada señal de control a generar (c_0, c_1, ..., c_{m-1}) y por cada bit del registro de estado (Q_0, Q_1, ..., Q_{n-1}).

2) En el array AND para cada estado S_i de la tabla de estados (ver Figura 5.5) se utiliza un número de puertas AND igual al número de estados que se pueden alcanzar desde S_i. Así, por ejemplo, para el estado S_0 se emplean dos puertas AND (cuando S_0 es el estado presente, el próximo estado puede ser S_0 ó S_{00}) mientras que para el estado S_{00} se utiliza una única puerta AND (cuando S_{00} es el estado presente, el próximo estado es siempre S_1). Si existe más de un próximo estado, éste estará determinado por el valor de las variables de condición (s_1 o s_0) y/o las señales externas (I).

La Figura 5.39 muestra la unidad de control del multiplicador implementada con este esquema (se utiliza el modelo de Moore), donde cada columna del array OR genera la función de transición de estados y de salida para la combinación (*estado_presente, entrada*) que se indica en la misma columna del array AND. La programación de conexiones de los cinco estados S_0, S_{00}, S_1, S_{10} y S_2 es la siguiente:

1) *Estado* S_0. En este estado se distinguen dos casos. El primero se produce cuando $I = 0$, con lo que las conexiones de la primera columna hacen que la puerta A_1 genere salida 1 (se conecta $\overline{Q}_2$, $\overline{Q}_1$ y $\overline{Q}_0$ a sus entradas). El estado siguiente sigue siendo $S_0 = (000)$ por lo cual la salida de A_1 no se conecta a ninguna de las puertas OR de salida O_3, O_2 y O_1 lo que indica que el próximo estado será otra vez S_0. Cuando $I = 1$, las conexiones de la segunda columna hacen que A_2 genere salida 1. El estado siguiente es $S_{00} = (001)$ por lo que la salida de A_2 se conecta a la entrada de O_1. Cuando el sistema se encuentra en el estado S_0 no se activa ninguna señal de control.

2) *Estado* S_{00}. Las conexiones de la tercera columna hacen que la puerta A_3 genere salida 1 (se conecta $\overline{Q}_2$, $\overline{Q}_1$ y Q_0 a sus entradas). El estado siguiente ha de ser $S_1 = (010)$ por lo cual se marcará la conexión entre la salida de A_3 y la fila de la puerta O_2. Además en S_{00} se deben activar las señales de control c_3 y c_0, y por tanto se ha de marcar la conexión entre la salida de A_3 y las filas de las puertas O_4 y O_7.

3) *Estado* S_1. En este estado se distinguen dos casos. El primero se produce cuando $s_0 = 0$, con lo que las conexiones de la cuarta columna hacen que la puerta A_4 genere salida 1 (se conecta $\overline{Q}_2$, Q_1 y $\overline{Q}_0$ a sus entradas). El estado siguiente es $S_2 = (100)$ por lo cual la salida de A_4 aparece conectada a la entrada de la puerta O_3. Cuando $s_0 = 1$, las conexiones de la quinta columna hacen que A_5 genere salida 1. El estado siguiente es $S_{10} = (011)$ por lo que la salida de A_5 se conecta a las entrada de O_1 y O_2. Cuando el sistema se encuentra en el estado S_1 no se activa ninguna señal de control.

4) *Estado* S_{10}. Las conexiones de la sexta columna hacen que la puerta A_6 genere salida 1 (se conecta $\overline{Q}_2$, Q_1 y Q_0 a sus entradas). El estado siguiente ha de ser $S_2 = (100)$ por lo cual se marcará la

conexión entre la salida de A_6 y la fila de la puerta O_3. Además en S_{10} se debe activar la señal de control c_1, y por tanto se ha de marcar la conexión entre la salida de A_6 y la fila de la puerta O_6.

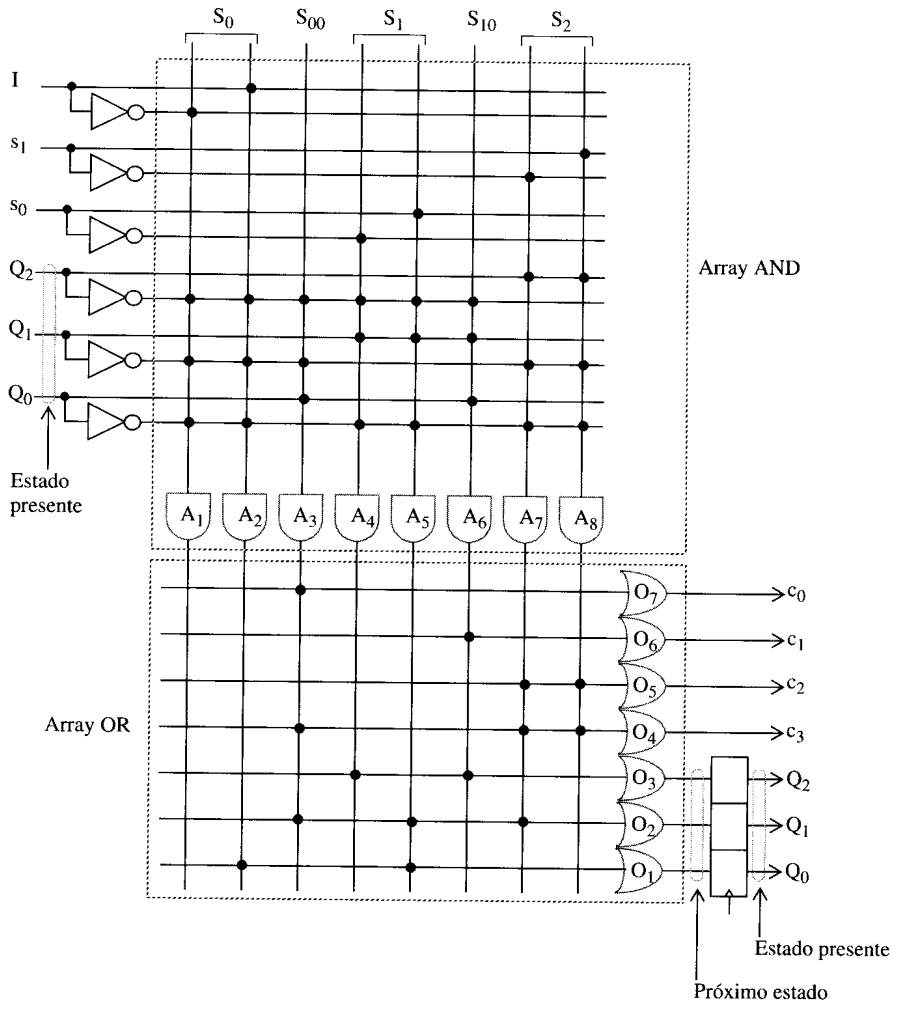


Figura 5.39: Realización de la unidad de control del multiplicador con un PLA

5) *Estado* S_2. En este estado se distinguen dos casos. El primero se produce cuando $s_1 = 0$, con lo que las conexiones de la séptima columna hacen que la puerta A_7 genere salida 1 (se conecta Q_2, $\overline{Q}_1$ y $\overline{Q}_0$ a sus entradas). El estado siguiente es $S_1 = (010)$ por lo cual la salida de A_7 aparece conectada a la entrada de la puerta O_2. Cuando $s_1 = 1$, las conexiones de la octava columna hacen que A_8 genere salida 1. El estado siguiente es $S_0 = (000)$ por lo cual la salida de A_1 no se conecta a ninguna de las puertas OR de salida O_3, O_2 y O_1 lo que indica que el próximo estado será S_0.

Además en S_2 se deben activar las señales de control c_2 y c_3, y por tanto se ha de marcar la conexión entre las salidas de A_7 y A_8 y las filas de las puertas O_4 y O_5.

Si el número de variables de condición es grande se pueden emplear estrategias similares a las que se han utilizado en el caso de las memorias ROM.

5.5.11 Resumen del procedimiento de diseño a nivel de registro

Como ya se ha señalado, en la síntesis de sistemas digitales a nivel de transferencia de registro no existe ningún método formal de diseño. Sin embargo, se puede utilizar una cierta *metodología heurística* que se ha ido presentando en las secciones anteriores. En esta sección, a modo de resumen, se enumeran los pasos que constituyen el procedimiento de diseño a nivel de registros y se concreta en el ejemplo del multiplicador binario que se ha utilizado a lo largo del tema:

1) *Definir el comportamiento del sistema digital* mediante un conjunto S de *secuencias de operaciones de transferencias entre registros (algoritmo)*. Este paso implica un proceso análogo a la codificación de un programa en lenguaje ensamblador. En la Figura 5.18 se muestra un ejemplo de la multiplicación de dos números binarios sin signo mediante el algoritmo de lápiz y papel mejorado y en la Figura 5.19 se da una descripción formal del algoritmo.

2) *Analizar el algoritmo* para determinar los tipos de componentes a emplear y el número de componentes de cada tipo. En este paso se definen los recursos de cálculo que contiene la unidad de procesamiento. Dado el algoritmo, la identificación de los componentes es un proceso bastante directo. Esto supone elegir una opción entre soluciones alternativas con diferentes compromisos en cuanto a su coste y velocidad.

3) *Construir un diagrama de bloques del sistema digital*, usando los componentes especificados en el paso 2. Elegir las interconexiones entre componentes de tal modo que se verifique que son posibles todas las transferencias de información expresadas en el algoritmo y que se satisface el criterio coste/prestaciones especificado en el diseño.

   En la Figura 5.20 se muestra el diagrama de bloques del multiplicador en el que se identifica la *unidad de procesamiento* y la *unidad de control* y la intercomunicación entre ambas a través de las señales de condición y las señales de control.

4) *Diseñar la unidad de control* de la máquina que genera las señales de control identificadas en el paso anterior, en el orden especificado por el algoritmo de transferencia de registros.

   Es posible emplear diferentes esquemas para realizar la unidad de control. Estas alternativas van desde arquitecturas muy rígidas y que hay que rediseñar desde el principio si se introduce alguna pequeña modificación en el diseño, hasta concepciones más flexibles y fácilmente modificables.

5) *Comprobar el funcionamiento del diseño resultante.*

Sección 5-6 Conclusiones

El problema central con el que se enfrenta el diseñador de un sistema digital es el de concebir estructuras que utilicen los componentes que hay disponibles y que realicen el conjunto de operaciones especificadas a un coste mínimo. A nivel abstracto esto significa *transformar la descripción de una conducta dada en una estructura apropiada.* Para conseguir este objetivo existen diferentes métodos descriptivos. En la Tabla 5.12 se resumen los más utilizados así como la función que realizan.

Método	Descripción
Diagrama de bloques	Estructura del sistema
Diagrama de flujos	Conducta del sistema
Lenguaje de descripción	Conducta del sistema

Tabla 5.12: Métodos de descripción de un sistema digital

Tanto con fines de descripción como de diseño, los sistemas digitales se suelen dividir en dos partes: *unidad de control* y *unidad de procesamiento.* Desde otro punto de vista, el diseño de un sistema digital puede considerarse a diferentes niveles de detalle o complejidad: *nivel de puertas lógicas, nivel de transferencia de registros* y *nivel de procesador.*

Cada nivel viene determinado por sus componentes elementales y por las unidades de información que maneja (el bit a nivel de puertas lógicas, la palabra a nivel de registro y el bloque de palabras a nivel de procesador). La arquitectura de un computador se describe a nivel de transferencia de registro o a nivel de procesador. El nivel de puertas lógicas es el nivel de diseño lógico clásico, que se caracteriza por poseer una teoría bien desarrollada (la teoría de la conmutación) basada en el álgebra booleana.

A nivel de transferencia de registro o a nivel de procesador no existe ninguna teoría equivalente, por lo que se suelen utilizar técnicas de diseño heurísticas. Las componentes del nivel de registro incluyen módulos combinacionales tales como puertas de palabras, multiplexores, decodificadores, sumadores, ROM's, PLA's etc y módulos secuenciales como registros, registros de desplazamiento, contadores, etc. Todos estos componentes cumplen la propiedad de ser fácilmente ampliables para permitir un mayor número de entradas.

El primer paso en el diseño a nivel de registro es construir una descripción formal de la conducta deseada. A partir de esta descripción se determinan las componentes que se necesitan, las conexiones (camino de datos) de la unidad de procesamiento y se construye su diagrama de bloques. A continuación se definen las señales de control que son necesarias y se identifican los puntos de control en la unidad de procesamiento. Finalmente se diseña la unidad de control que genera las señales de control en la secuencia correcta.

Sección 5-7 Problemas

1) Diseñar los sistemas digitales que realizan las siguientes transferencias de registro:

a) Si x es impar entonces a ← b + c
si no a ← b - c

b) Si x ≠ 0 entonces a ← a - 1
si no a ← a + 1

Suponer que *a, b, c* y *x* son registros de 8 bits.

2) La unidad de control de un sistema digital dispone de tres entradas s_0, s_1 y s_2 y su diagrama de estados se corresponde con el que se muestra en la Figura 5.40.

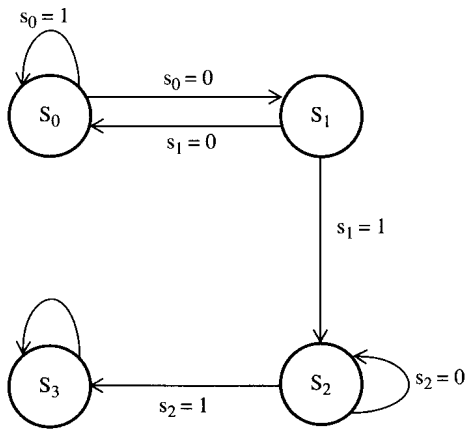


Figura 5.40: Diagrama de estados de la unidad de control del Problema 2

Diseñar la unidad de control utilizando elementos de memoria tipo *D* y puertas lógicas (ver apartado 5.5.5).

3) Repetir el problema 2 utilizando un registro de secuencia y un decodificador como componentes básicos (ver apartado 5.5.6).

4) Repetir el problema 2 utilizando un elemento de memoria tipo *D* por estado (ver apartado 5.5.7).

5) Repetir el problema 2 utilizando una memoria ROM y un registro de estado. Discutir las diferentes alternativas del diseño (ver apartado 5.5.8).

6) Repetir el problema 2 utilizando un contador y un decodificador como componentes básicos (ver apartado 5.5.9).

7) Repetir el problema 2 utilizando un array lógico programable (PLA) y un registro de estado (ver apartado 5.5.10).

8) En la Figura 5.41 se muestra la unidad de procesamiento de un sistema digital

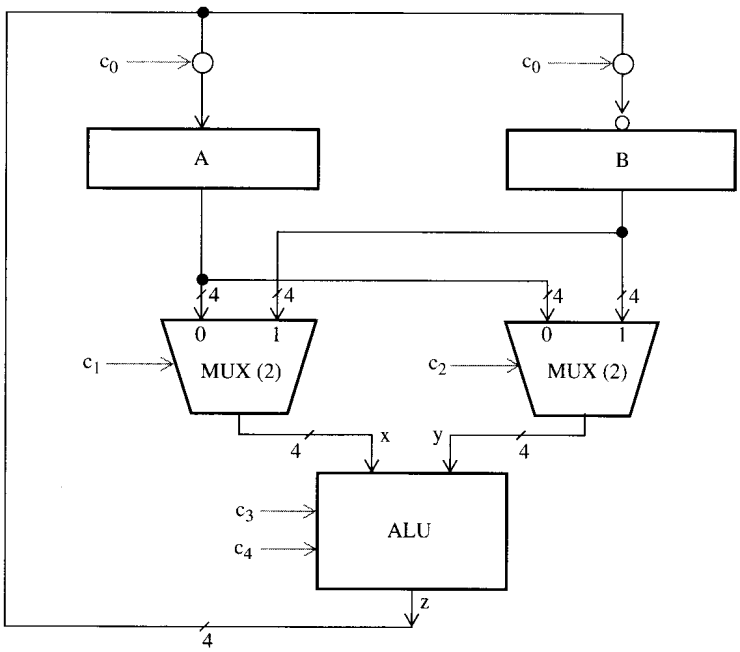


Figura 5.41: Unidad de procesamiento del Problema 8

En la Tabla 5.13 se da el significado de los distintos puntos de control. Cada palabra de control debe especificarse de acuerdo con el formato: $c_4 c_3 c_2 c_1 c_0$. Por ejemplo:

$$c_4 c_3 c_2 c_1 c_0 = 0\,0\,1\,0\,1 \quad \Rightarrow \quad A \leftarrow A + B$$

Responder a las siguientes cuestiones escribiendo la(s) palabra(s) de control adecuada(s).

a) ¿Cómo se puede poner el registro B a cero? (Sugerir al menos dos posibilidades). No se dispone de una entrada directa para poner a cero el registro.

b) Determinar una secuencia de control que intercambie los contenidos de los registros A y B ($A \leftarrow B$ y $B \leftarrow A$).

Señal de Control	Operación controlada
$c_0 = 0$	B ← z
$c_0 = 1$	A ← z
$c_1\ c_2 = 0\ 0$	x = A y = A
$c_1\ c_2 = 0\ 1$	x = A y = B
$c_1\ c_2 = 1\ 0$	x = B y = A
$c_1\ c_2 = 1\ 1$	x = B y = B
$c_3\ c_4 = 0\ 0$	z = x + y
$c_3\ c_4 = 0\ 1$	z = x - y
$c_3\ c_4 = 1\ 0$	z = x AND y
$c_3\ c_4 = 1\ 1$	z = x ⊕ y

Tabla 5.13: Señales de control de la unidad de procesamiento del problema 8

9) Sea el siguiente algoritmo:

```
1:  Declaración de registros A[8], B[8], R[8];
2:  Inicio    A ← 0;
3:            B ← 00001111;
4:  Bucle     A ← A + B;
5:            B ← B - 1;
6:            if B <> 0 then go to Bucle;
7:            R ← A;
8:            Parar;
```

¿Qué operación se ejecuta? Diseñar un sistema digital (unidad de procesamiento y unidad de control) que realice este algoritmo. Discutir las diferentes alternativas de diseño.

10) El siguiente algoritmo describe una determinada operación de un sistema digital:

```
1:  Declaración de registros A[8], B[8], R[8], I[1];
2:  Declaración de buses bus-entrada[8], bus-salida[8];
3:  Inicio    if I = 0 then go to Inicio;
4:            A ← bus-entrada;
5:            B ← bus-entrada;
6:            B ← A + B;
7:            if B(7) = 0 then go to Par;
8:            B ← B̄;
9:            B ← B + 1;
10:           go to Fin;
11: Par       A ← A + 1;
```

```
12:  Fin     R ← A + B, Bus-Salida ← A + B;
13:          go to Inicio;
```

a) Si los valores iniciales cargados en los registros *A* y *B* desde el bus-entrada son:

$$A = (0\,1\,1\,0\,1\,0\,1\,0) \qquad B = (1\,1\,0\,1\,1\,0\,1\,0)$$

determinar los valores de *A*, *B* y *R* después de la ejecución del algoritmo.

b) Diseñar una unidad de procesamiento que resulte adecuada para la ejecución del algoritmo.

c) Diseñar la unidad de control del sistema utilizando los procedimientos vistos en la Sección 5-5.

11) Para el diagrama de estados que se muestra en la Figura 5.42, diseñar la unidad de control en los dos casos siguientes:

a) Se dispone de una variable de condición por estado

b) Se dispone de dos variables de condición por estado

Efectuar la síntesis con los siguientes componentes:

1) Elementos de memoria tipo *D* y puertas lógicas

2) Un contador y un decodificador

3) Una memoria ROM y un registro de estado

4) Un array lógico programable (PLA) y un registro de estado

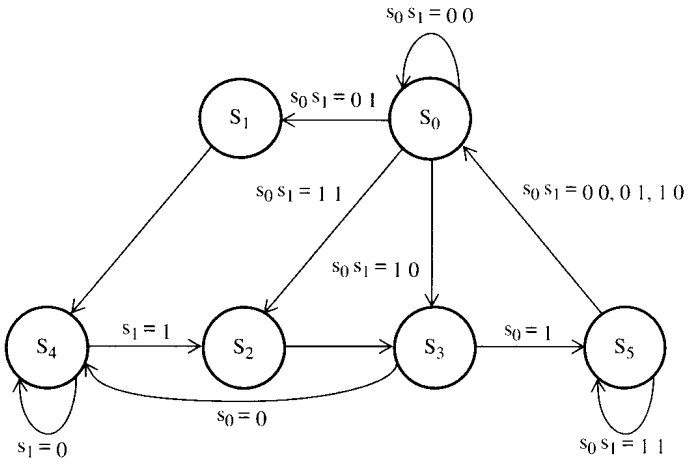


Figura 5.42: Diagrama de estados de la unidad de control del problema 11

12) Diseñar un sistema digital que cuente el número de ceros de una palabra de datos de entrada y devuelva al final el resultado.

13) Obtener el diagrama ASM de un elemento de memoria *JK*.

14) Obtener el diagrama ASM de un multiplicador de desplazamiento y suma que sume el multiplicando (M) al producto parcial (PP) si el bit menos significativo del multiplicador (m) es 1 y desplace el producto parcial y el multiplicador un bit a la derecha.

15) Obtener el diagrama ASM de un contador que cuenta en módulo 4 si la señal x es igual a 0 y en módulo 7 si es igual a 1.

16) Un método rápido de obtener, de forma serie, el complemento a 2 de un número es el siguiente: se van examinando los bits del número a complementar de derecha a izquierda y se dejan inalterados hasta la aparición del primer 1, éste se deja inalterado y a partir de ese instante se complementan todos los bits. Obtener el diagrama ASM de un complementador a 2 serie.

17) Obtener el diagrama ASM de un contador síncrono de 3 bits que dispone de una entrada de control m. Cuando $m = 0$ el contador sigue de forma repetida la secuencia binaria 000, 001, 011, 010, 100, 101, 110, 111. Si $m = 1$, el contador avanza a través de la siguiente secuencia de código Gray 000, 001, 011, 010, 110, 111, 101, 100 de forma repetida.

18) En la Figura 4.40 del tema 4 se da el diagrama de flujo del algoritmo de Booth para la multiplicación de dos números enteros representados en complemento a 2. Diseñar un sistema digital (unidad de procesamiento + unidad de control) que realice este algoritmo. (Suponer que los registros tienen una longitud de 8 bits)

19) En la Figura 4.53 del tema 4 se da el diagrama de flujo del algoritmo de división de dos números enteros utilizando el método de restauración. Diseñar un sistema digital (unidad de procesamiento + unidad de control) que realice este algoritmo. (Suponer que los registros tienen una longitud de 8 bits)

20) En la Figura 4.54 del tema 4 se da el diagrama de flujo del algoritmo de división de dos números enteros utilizando el método de no restauración. Diseñar un sistema digital (unidad de procesamiento + unidad de control) que realice este algoritmo. (Suponer que los registros tienen una longitud de 8 bits)

21) Diseñar un sistema digital (unidad de procesamiento + unidad de control) que sea capaz de calcular la raíz cuadrada por defecto de un número entero. Es decir, dado un número x de n bits, calcular z tal que:

$$z^2 \leq x \leq (z+1)^2$$

Suponer que los registros tienen una longitud de 8 bits.

22) En la Figura 5.32 se muestra una realización de la unidad de control basada en una memoria ROM que utiliza un multiplexor para seleccionar 1 de q condiciones posibles (selección por estado). Generalizar este esquema cuando el próximo estado depende de p condiciones ($p < q$).

23) Diseñar un controlador de semáforos simplificado que conmute las luces en un cruce cuando una calle norte-sur (NS) se cruce con una calle este-oeste (EO). La entrada del controlador es el pulsador *PASAR* que lo presionan los peatones cuando quieren atravesar la calle. La salida son dos señales *NS* y *EO*. Cuando *NS* o *EO* son 0, se enciende la luz roja y cuando son 1, se enciende la luz verde. Si no hay peatones, $NS = 0$ y $EO = 1$ durante 1 minuto, y después $NS = 1$ y $EO = 0$ durante otro minuto y así sucesivamente. Cuando se pulsa el botón *PASAR*, *NS* y *EO* se ponen a 1 durante 1 minuto, siempre que el minuto actual haya concluido. Después de esto, las señales *NS* y *EO* continúan alternándose. Para este controlador de semáforos:

a) Obtener el diagrama de estados del controlador de semaforos.

b) Codificar los estados.

c) Obtener un circuito lógico.

24) Diseñar un sistema digital (unidad de procesamiento + unidad de control) que sea capaz de calcular las expresiones siguientes:

a) $r = \sum_{i=0}^{n} a_i x_i$

b) $r = \sum_{i=0}^{n} a_i x_i + b_i$

Tema 6 Diseño del procesador

La unidad central de proceso (CPU) es un elemento esencial de cualquier computador ya que tiene como misión ejecutar las instrucciones de un programa. Las instrucciones se procesan de forma secuencial, leyéndose de posiciones consecutivas de memoria (tras ejecutar la instrucción que se encuentra en la dirección x se ejecuta la instrucción que está en la dirección $x+1$[1] y así sucesivamente). Este esquema que se conoce como *secuenciamiento implícito* sólo puede ser modificado mediante instrucciones de bifurcación o salto. En este tipo de instrucciones se debe especificar la dirección de la siguiente instrucción que se va a ejecutar o un mecanismo para calcularla. La CPU como cualquier sistema digital y según el modelo de Glushkov (ver apartado 5.3.3) se puede descomponer en los dos bloques siguientes:

a) *Unidad de procesamiento o ruta de datos.* Esta unidad se encarga de realizar todas las operaciones necesarias para ejecutar una instrucción. Es un sistema secuencial gestionado por la secuencia de señales de control que se generan en la unidad de control. También se ocupa de generar la información necesaria para que la unidad de control conozca el estado de ejecución de cada instrucción y pueda decidir así cuál es el correcto secuenciamiento de las señales de control para ejecutar el programa.

b) *Unidad de control.* La función de la *unidad de control* de un computador es la búsqueda de las instrucciones en memoria, su interpretación y la generación, en los instantes adecuados, de las señales de control necesarias para ejecutar la operación especificada por cada instrucción. En este proceso se pueden distinguir dos aspectos esenciales: el *secuenciamiento* y la *interpretación* de las instrucciones. Fundamentalmente la unidad de control necesita conocer qué tipo de instrucción se está ejecutando. En el caso de que se trate de una instrucción de bifurcación debe disponer de la información necesaria para saber si debe romper el secuenciamiento implícito del programa o no.

El diseño de la unidad de control de una CPU no difiere sustancialmente del diseño de la unidad de control del multiplicador binario que se consideró en el tema anterior, y por lo tanto la metodología que se

1. En general, si la instrucción necesita p palabras de memoria, el incremento que debe sufrir el contador de programa (PC) para pasar a la ejecución de la siguiente instrucción será: $PC \leftarrow PC + p$. En este tema y por simplicidad se supone que $p = 1$.

desarrolló es perfectamente válida en este caso. La diferencia principal estriba en que el número de líneas de control que se dispone ahora puede ser mucho mayor y las operaciones a realizar más complicadas. Una CPU puede contener varios cientos de líneas de control que hacen que el diseño de la unidad de control resulte bastante complejo.

En la Figura 6.1 se muestra un diagrama de bloques de las interconexiones entre la unidad de procesamiento, la unidad de control y la memoria. Además del intercambio de señales descrito entre la unidad de control y la unidad de procesamiento, ésta última debe comunicarse con la memoria del computador a través de los buses de datos y direcciones para poder acceder a las instrucciones y a los datos. Por su parte la unidad de control se encarga de gestionar la señal de lectura/escritura en memoria a través del bus de control.

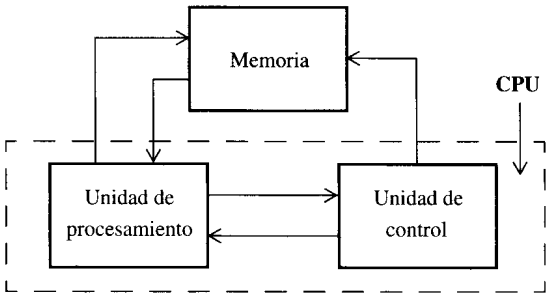


Figura 6.1: Diagrama de bloques CPU-Memoria

La unidad de control de un procesador dirige la ejecución de un programa. Tiene en cuenta el estado presente para controlar su evolución y tomar decisiones con respecto a las nuevas operaciones que serán iniciadas. Como toda máquina secuencial, la unidad de control puede ser descompuesta en tres bloques:

- Un registro de memorización del estado presente
- Un bloque de cálculo del próximo estado
- Un bloque de cálculo de las salidas (las microórdenes enviadas a la unidad de procesamiento)

Dos realizaciones son posibles para esta unidad de control:

1) *Realización cableada.* Se construye con puertas lógicas y su concepción obedece a las técnicas clásicas de diseño de circuitos secuenciales; los bloques de cálculo del próximo estado y de las salidas se realizan con sistemas lógicos combinacionales (ver Figura 6.2). En general son circuitos muy rápidos aunque complejos y costosos. Su gran desventaja es por tanto que cualquier modificación de las especificaciones del problema implica una alteración del sistema y eventualmente un nuevo diseño partiendo de cero. Por esta razón, así como por un mantenimiento más fácil, se prefiere generalmente una realización microprogramada y no una realización cableada.

2) *Realización microprogramada.* El bloque de cálculo de las señales de salida y una parte del bloque de cálculo del próximo estado se sustituyen por una memoria que almacena el estado de las señales de control en cada periodo de cada instrucción. Esta memoria recibe el nombre de *memoria de control*. A cada palabra, que define un periodo de una instrucción se la denomina

microinstrucción. El conjunto ordenado de microinstrucciones que realizan una instrucción se llama *microprograma*. El registro de estado presente se convierte ahora en un registro de dirección para esta memoria. Su estructura es la que se muestra en la Figura 6.3. El control está completamente centralizado: toda la información de control se encuentra en la memoria de control, contrariamente a lo que pasa en una realización cableada, donde esta información está dispersa en un conjunto de sistemas lógicos. La realización microprogramada permite generalizar el diseño de la unidad de control basado en la utilización de memorias ROM que se consideró en el tema 5. Las funciones de transición de estado y de salida se sintetizan mediante tablas de consulta que están almacenadas en la memoria de control.

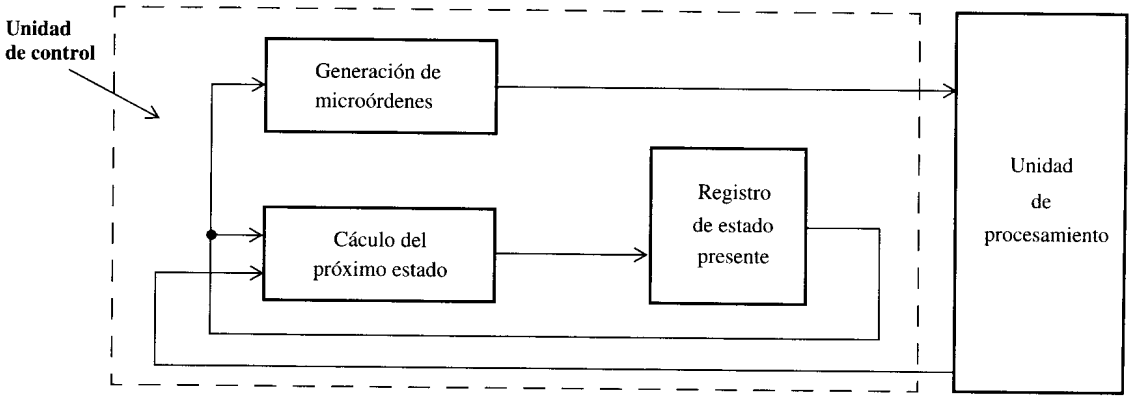


Figura 6.2: Estructura de una unidad de control con lógica cableada

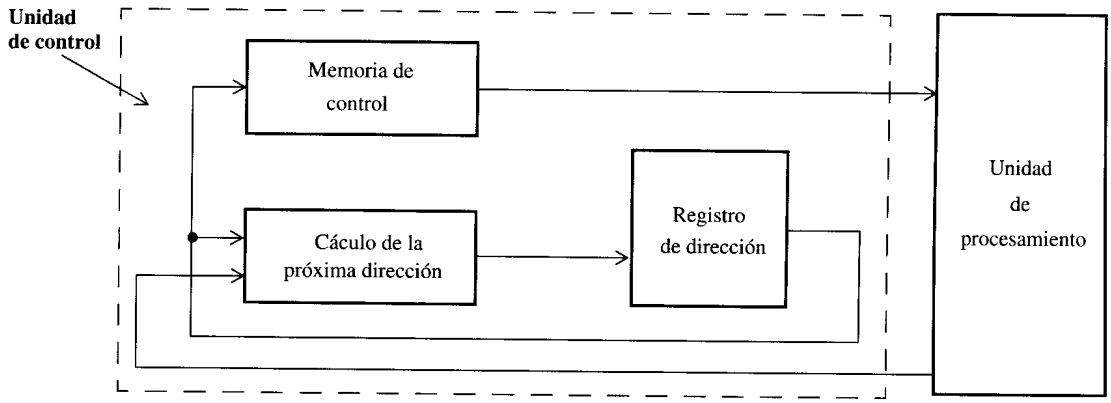


Figura 6.3: Estructura de una unidad de control microprogramada

La unidad de procesamiento o ruta de datos de un procesador puede ser descrita mediante una lista de todas sus *microoperaciones*. Esta lista, que inclusive para los grandes procesadores no es superior a 150 microoperaciones, permite ver todos los operandos, todos los operadores y todos los caminos de interco-

nexión de la unidad de procesamiento, es decir su estructura.

Desde el punto de vista más elemental, todas y cada una de las microoperaciones que hay que realizar para la ejecución de cualquier instrucción máquina se pueden clasificar en dos grandes grupos: *microoperaciones de transferencia* y *microoperaciones de proceso* (ver Figura 6.4). En las primeras, la información viaja entre dos elementos de almacenamiento (registros o posiciones de memoria), uno origen y otro destino. En las segundas, la información origen se transforma al pasar a través de un operador en su camino hacia el destino.

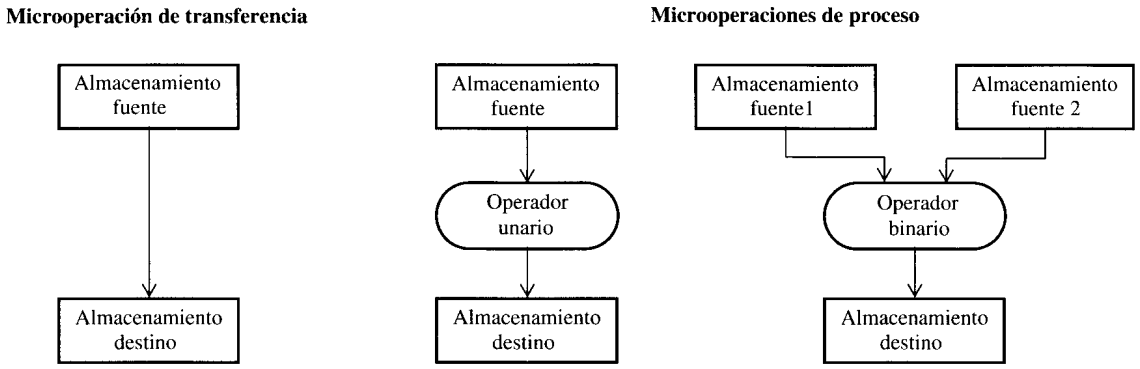


Figura 6.4: Microoperaciones básicas

En este tema se estudia la unidad de control de un computador realizada con lógica cableada. En primer lugar se analiza el *repertorio de instrucciones* de los procesadores y se procede a una clasificación de los mismos según el número de direcciones que contenga el formato de sus instrucciones.

La siguiente sección está dedicada a describir los *modos de direccionamiento* de un computador que permiten calcular de forma no ambigua la dirección real o efectiva donde se encuentran los operandos de la instrucción y dónde hay que almacenar el resultado que produce.

A continuación se analiza de forma detallada el *ciclo de ejecución de una instrucción*. La ejecución de una instrucción siempre conlleva realizar la misma secuencia de pasos independientemente del repertorio de instrucciones específicas que posea el procesador y de los campos y modos de direccionamiento que se hayan definido. De forma genérica, el ciclo de ejecución de una instrucción se puede subdividir en cuatro fases: 1) Fase de búsqueda de la instrucción, 2) Fase de decodificación de la instrucción, 3) Fase de búsqueda de los operandos y 4) Fase de ejecución de la instrucción.

Se presentan también las etapas que son necesarias para realizar de una forma sistemática el diseño de un procesador genérico con lógica cableada. Finalmente en la última sección del tema se procede a realizar el diseño de SIMPLE1 que es un procesador elemental y académico que permite mostrar de forma sencilla los conceptos que se han ido introduciendo. El problema se especifica dando las características estructurales de SIMPLE1, un procesador ficticio extraordinariamente simplificado, cuya única finalidad es mostrar paso a paso su diseño con lógica cableada.

Sección 6-1

Repertorio de instrucciones

Un procesador se diseña con el objetivo de ejecutar secuencias de instrucciones que constituyen un *programa*. Cada una de las instrucciones lleva a cabo una parte elemental del procesamiento, como por ejemplo:

- Cargar un dato que está en la memoria en uno de los registros del procesador.
- Almacenar en la memoria un dato que está en uno de los registros del procesador.
- Realizar en la ALU del procesador una operación aritmética, lógica o de desplazamiento.
- Determinar dónde se encuentra la siguiente instrucción que se va a ejecutar.

El repertorio de instrucciones de un procesador dispondrá de un conjunto suficientemente amplio de ellas que pueden clasificarse según diferentes formatos. Las instrucciones del programa se interpretan en la unidad de control y se ejecutan en su unidad de procesamiento o ruta de datos.

Cada CPU tiene su propio y específico formato de instrucciones por lo que sería imposible describirlos todos y poco operativo presentar detalladamente algunos casos particulares. El objetivo de esta sección es dar una visión general de las principales alternativas, partiendo de las ideas básicas ya expuestas en el tema 1 (ver Ejemplo 1.2.2).

Una *instrucción* se puede ver como una cadena de bits que se agrupan en *campos* con tamaños diferentes. Si se desea tener el máximo número de instrucciones con la mínima longitud de palabra es conveniente un diseño óptimo del formato de instrucciones. Esto lleva generalmente a considerar tres características que suelen estar presentes en los formatos de instrucciones de la mayoría de los computadores.

1) La longitud de los distintos campos no es igual para todas las instrucciones.
2) No necesariamente todos los campos tienen que estar presentes en todas las instrucciones.
3) Como consecuencia de lo anterior las instrucciones del repertorio de una máquina concreta pueden tener diversos tamaños, es decir unas instrucciones pueden ocupar un solo byte, otras dos, etc.

Cada formato de instrucción (ver Figura 6.5) contiene un campo denominado *código de operación* (*código_op*) que especifica la función que realiza la instrucción y un *campo de dirección* que especifica dónde localizar los operandos (o algunas veces los propios operandos) que son necesarios para dicha operación.

Para definir cada instrucción, normalmente se emplea una notación que especifica en primer lugar el *código_op* seguido por las direcciones que sean necesarias. Por ejemplo, considérese la instrucción de la operación $u = v \times w$, con u, v y w almacenados respectivamente en las posiciones U, V y W. Para representar

esta instrucción se podría utilizar uno de los dos formatos siguientes: *Mult U, V, W* ó $M[U] \leftarrow M[V] \times M[W]$, donde M designa a un array de memoria y u, v y w son respectivamente los elementos $M[U]$, $M[V]$ y $M[W]$ de dicho array (los símbolos entre corchetes indican los índices del array). Es también posible que las variables estén almacenadas en un banco de registros de datos que el procesador tiene en su unidad de procesamiento o ruta de datos, en cuyo caso se podría escribir: *Mult R1, R2, R3* ó $R[1] \leftarrow R[2] \times R[3]$, que indica que las direcciones $R1$, $R2$ y $R3$ son direcciones del banco de registros.

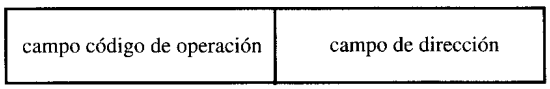


Figura 6.5: Formato básico de una instrucción

Además de los campos *código_op* y *dirección*, una instrucción puede incluir dos campos adicionales: un *campo de tipo de instrucción* y un *campo de modo*.

Los tipos de instrucciones se pueden clasificar en unos de los cinco grupos siguientes:

1) *Instrucciones de transferencia de datos.* Se emplean para transferir (copiar) datos desde una posición (de memoria o un registro) a otra posición (de memoria o un registro). En este grupo se incluyen instrucciones tales como *cargar*, *almacenar* y *mover*. Además de considerar los movimientos internos (que consisten en trasladar datos entre la memoria y los registros accesibles al programador), se consideran también las comunicaciones con los dispositivos periféricos.

2) *Instrucciones aritméticas, lógicas y de comparación.* Las instrucciones aritméticas efectúan las operaciones aritméticas básicas tales como incrementar, decrementar, sumar, restar, multiplicar y dividir. Las instrucciones lógicas realizan operaciones binarias sobre palabras que están almacenadas en registros o en posiciones de memoria. Son útiles para manipular bits individuales o grupos de bits que representan la información codificada en binario. Las instrucciones lógicas consideran de forma independiente cada bit del operando y lo tratan como una variable booleana a la que puede aplicar las operaciones NOT, AND, OR y OR-exclusiva. Las instrucciones de comparación hacen una resta de los dos operandos (aunque esta diferencia no se retiene) que puede originar una bifurcación condicional, cambiar el contenido de un determinado registro o poner a cero o a uno ciertos bits del registro de estado.

3) *Instrucciones de desplazamiento.* Las instrucciones de desplazamiento son operaciones en las cuales los bits del operando se mueven hacia la izquierda o hacia la derecha. Se pueden especificar operaciones de desplazamiento lógico o aritmético y de rotación. En la sección 4-9 se han presentado ya de forma detallada todas las opereraciones de desplazamiento.

4) *Instrucciones de transferencia de control.* Controlan la secuencia de ejecución de instrucciones. Además de las transferencias de control incondicional se encuentran en este grupo las condicionadas al valor que tengan determinados indicadores (por ejemplo "bifurcación si cero", "bifurcación si no cero", "bifurcación si negativo", etc).

5) *Instrucciones de gobierno.* Son las que actúan sobre la CPU y sobre unidades de control periféricas, bien sea para modificar su estado o para comprobarlo. Generalmente suelen ser

"instrucciones privilegiadas" y a menudo su uso está reservado al sistema operativo como en las llamadas al sistema y en las instrucciones de gestión de la memoria.

El campo de modo especifica la forma de obtener la dirección efectiva de los datos a partir de la información que proporciona el campo de dirección. Por ejemplo, si el campo de modo especifica que la instrucción proporciona un direccionamiento indirecto, se tiene que el campo de dirección no indica la dirección real del operando sino una dirección de memoria donde se encuentra dicha dirección real (se designa de forma indirecta). Sea la siguiente instrucción: *Movei U, V* ($M[U] \leftarrow M[M[V]]$).

En este ejemplo debido al campo de modo de la instrucción (*Movei* indica que la instrucción de mover se realiza mediante un direccionamiento indirecto) $M[V]$ indica una dirección de memoria cuyo contenido es la dirección donde se encuentra almacenado el operando que se desea mover a la posición de memoria $M[U]$ y no el propio operando, como sería si la instrucción fuese *Move U, V* ($M[U] \leftarrow M[V]$).

Al definir el repertorio de instrucciones de un procesador, uno de los elementos claves que hay que considerar es el número de campos de dirección que poseen las instrucciones. Este número tiene una incidencia directa sobre la longitud de los programas y las prestaciones del procesador. Obviamente cuanto mayor sea el número de campos de dirección que tiene una instrucción, mayor será la longitud de palabra de las instrucciones pero en compensación los programas serán más pequeños ya que necesitarán menos instrucciones.

En el otro extremo, se tiene que cuando las instrucciones poseen menos campos de dirección tendrán menor tamaño lo que implica que se necesitará un programa más largo para efectuar la misma tarea. Hay pues que buscar un compromiso entre programas cortos con instrucciones complejas y programas largos con instrucciones sencillas. Una medida de las prestaciones del procesador que se puede tomar para establecer una comparación objetiva es el número total de accesos a memoria que se requieren.

De una parte cuando se utilizan instrucciones largas, el procesador realiza más accesos a memoria por instrucción (para poder leerla y obtener los operandos), sin embargo al ser éstas más complejas el programa necesitará un menor número de ellas. De otra si las instrucciones son más cortas se necesitan menos accesos a memoria por instrucción aunque el programa necesitará un mayor número de ellas. Un objetivo importante en el diseño del repertorio de instrucciones es *conseguir que por término medio el número de accesos a memoria que necesitan los programas sea mínimo*. A continuación se analizan las alternativas más usuales con un ejemplo sencillo que calcula la siguiente sentencia de asignación: $a := a \times b + c \times d \times e$.

6.1.1 Procesadores de tres direcciones

La mayoría de las operaciones aritméticas y lógicas se realizan sobre dos operandos y producen un resultado, por lo que resulta natural concebir instrucciones que incluyen *tres direcciones de memoria*: las dos de los operandos y una tercera para almacenar el resultado (ver Figura 6.6). Este formato se adoptó en algunos de los primeros computadores, en los que la capacidad de la memoria era pequeña y por tanto bastaba con pocos bits para los campos de dirección. De no ser así, la longitud del formato resulta ser excesivamente grande.

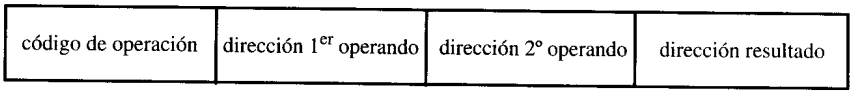


Figura 6.6: Formato de una instrucción de tres direcciones

En la Figura 6.7, se muestra que el cálculo de $a := a \times b + c \times d \times e$ podría especificarse mediante cuatro instrucciones, en las que se supone que las variables a, b, c, d y e más una variable temporal x están almacenadas en las posiciones A, B, C, D, E y X respectivamente:

1. *Mult A, A, B*; $(M[A] \leftarrow M[A] \times M[B])$
2. *Mult X, C, D*; $(M[X] \leftarrow M[C] \times M[D])$
3. *Mult X, X, E*; $(M[X] \leftarrow M[X] \times M[E])$
4. *Add A, A, X*; $(M[A] \leftarrow M[A] + M[X])$

Figura 6.7: Programa que calcula $a := a \times b + c \times d \times e$ en un procesador de tres direcciones

Este ejemplo revela el punto crítico de las instrucciones con tres operandos: requieren un número elevado de bits para codificarla. Si se supone que el tamaño de la memoria es de $256M$ palabras, cada campo de dirección de la instrucción precisaría de 28 bits de forma que una instrucción con tres direcciones necesitaría del orden de 90 bits (si se incluye también el campo de 6 bits de *código_op*). Si además se supone que las instrucciones con tres direcciones se ejecutan en un procesador de 32 bits con palabras de memoria también de 32 bits, cada instrucción necesitará tres palabras de memoria. Es decir, en estas circunstancias el procesador debe realizar tres accesos a memoria para traer una instrucción y tres más para ejecutarla (uno por cada uno de los datos que tiene que localizar). Con todo esto, para ejecutar el programa de la Figura 6.7, el procesador tendría que acceder doce veces a memoria para traer las cuatro instrucciones y otras doce más para acceder a los operandos y almacenar el resultado. En total se precisan pues veinticuatro accesos a memoria. En la Figura 6.8 se muestra el movimiento general de los operandos suponiendo que se encuentran almacenados en la memoria principal.

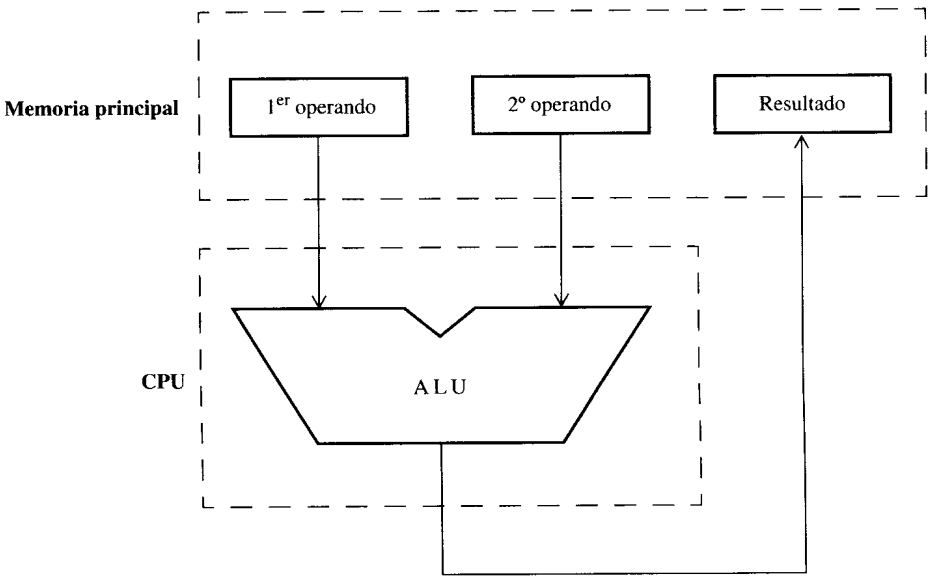


Figura 6.8: Flujo de datos en un procesador de tres direcciones en las instrucciones con referencia a memoria

Puede existir un cuarto campo de dirección que indica explícitamente la dirección donde se encuentra la siguiente instrucción que hay que ejecutar. De este modo, las instrucciones de transferencia de control o de bifurcación serían innecesarias, ya que todas las instrucciones incluirían explícitamente una bifurcación. En contraposición con el secuenciamiento implícito a este tipo de secuenciamiento se le conoce como *secuenciamiento explícito*. Este formato no se encuentra en los procesadores actuales (salvo en otro nivel que es el de la microprogramación tal como se verá en el tema 7).

6.1.2 Procesadores de dos direcciones

En las máquinas que utilizan *instrucciones con dos direcciones* el resultado puede depositarse en el lugar que inicialmente ocupaba uno de los operandos (el primero o el segundo según el diseño de la máquina). En la Figura 6.9 se muestra el formato de una instrucción de dos direcciones en la que se ha supuesto que el primer operando y el resultado comparten la misma posición de memoria.

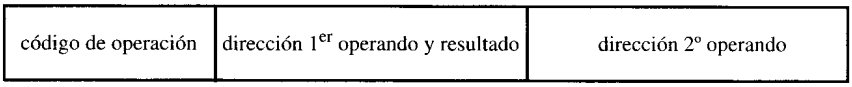


Figura 6.9: Formato de una instrucción de dos direcciones

En la Figura 6.10, se ve que el cálculo de $a := a \times b + c \times d \times e$ emplea cinco instrucciones. Como en el caso anterior se considera que cada dirección requiere 28 bits por lo que cada una de las instrucciones de dos direcciones ocupa ahora dos palabras de memoria en lugar de tres. Sin embargo se necesitan cinco instrucciones para calcular el mismo resultado. Por lo tanto el procesador debe hacer diez accesos a memoria para traer las instrucciones y catorce accesos para obtener los operandos y almacenar el resultado.

Con esta situación, el programa con instrucciones de dos direcciones tendría un tiempo de ejecución igual que el del programa con tres instrucciones. En este ejemplo concreto se compensa la mayor longitud del programa (cinco instrucciones frente a cuatro) con el menor numero de accesos a memoria que necesitan las instrucciones del procesador de dos direcciones para cargar los operandos.

1. *Mult A, B*; $(M[A] \leftarrow M[A] \times M[B])$
2. *Move X, C*; $(M[X] \leftarrow M[C])$
3. *Mult X, D*; $(M[X] \leftarrow M[X] \times M[D])$
4. *Mult X, E*; $(M[X] \leftarrow M[X] \times M[E])$
5. *Add A, X*; $(M[A] \leftarrow M[A] + M[X])$

Figura 6.10: Programa que calcula $a := a \times b + c \times d \times e$ en un procesador de dos direcciones

Conviene observar la forma en que se ha usado la instrucción *Move* para asignar el valor inicial de la variable temporal *X*. Al escribir el fragmento de programa de la Figura 6.10 se ha aprovechado el hecho de que la multiplicación y la suma son operaciones conmutativas.

Si en el ejemplo se hubiesen propuesto operaciones de división y resta se podría haber necesitado que se reordenasen partes de los cálculos o que se empleasen más instrucciones para calcular el resultado. En la Figura 6.11 se muestra el flujo de datos en un procesador de dos direcciones en las instrucciones con referencia a memoria.

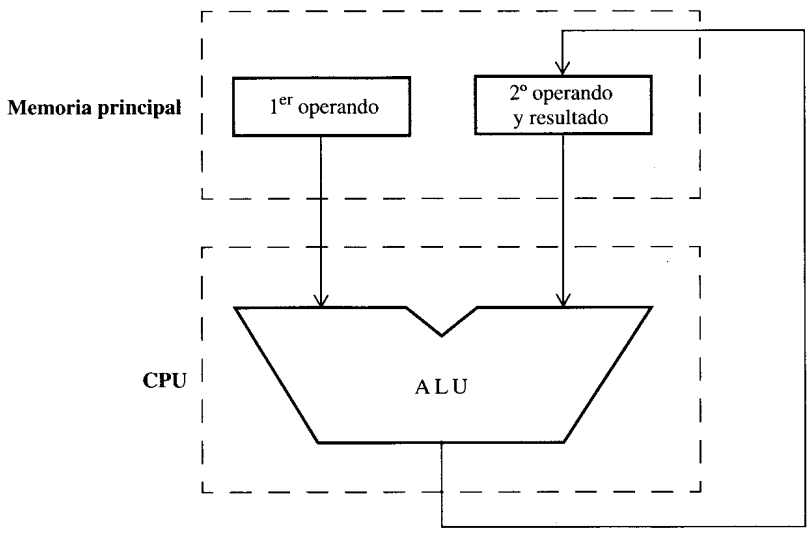


Figura 6.11: Flujo de datos en un procesador de dos direcciones en las instrucciones con referencia a memoria

6.1.3 Procesadores de una dirección (procesadores con acumulador)

En las máquinas que utilizan *instrucciones con una dirección* (o con acumulador), el procesador dedica un registro específico de su ruta de datos, que se conoce como *acumulador* (*Acum*), a la función de servir como operando destino. Por este motivo sólo es necesario especificar un operando de forma explícita en cada instrucción: la dirección del otro operando fuente (ver Figura 6.12).

$$\text{Acum} \leftarrow \text{F(Acum, M[dir])} \quad \text{ó} \quad \text{Acum} \leftarrow \text{M[dir]}$$

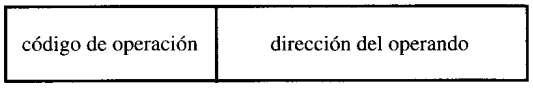


Figura 6.12: Formato de una instrucción de una dirección

Hay sin embargo una excepción a la regla de que el acumulador sea el operando destino. En la instrucción *Store* (*almacena*), el acumulador es el operando fuente y la dirección explícita especifica el operando destino.

$$\text{M[dir]} \leftarrow \text{Acum}$$

En la Figura 6.13, se observa que el cálculo de $a := a \times b + c \times d \times e$ en un procesador con acumulador emplea ocho instrucciones. Con las mismas hipótesis anteriores, cada instrucción sólo necesita ahora una palabra de memoria, lo que significa que el procesador accede ocho veces a memoria para traer las instrucciones y otras ocho para obtener los operandos y almacenar el resultado. En total se precisan pues dieciseis accesos a memoria. Según esto, éste fragmento de programa requiere un tiempo menor que cualquiera de sus predecesores. La razón para que esto suceda es que el acumulador permite guardar

temporalmente los operandos y los resultados para que el procesador no tenga necesidad de acceder tanto a memoria.

```
1. Load A;      (Acum ← M[A])
2. Mult B;      (Acum ← Acum × M[B])
3. Store A;     (M[A] ← Acum)
4. Load C;      (Acum ← M[C])
5. Mult D;      (Acum ← Acum × M[D])
6. Mult E;      (Acum ← Acum × M[E])
7. Add A;       (Acum ← Acum + M[A])
8. Store A;     (M[A] ← Acum)
```

Figura 6.13: Programa que calcula $a := a \times b + c \times d \times e$ en un procesador con acumulador

Se constata que en este caso no existe ninguna variable temporal explícita (X) ya que el acumulador actúa como la memoria temporal necesaria. Sin embargo, el acumulador sólo puede actuar como una memoria temporal simple y si se calculasen expresiones más complicadas se podrían necesitar memorias temporales explícitas. En la Figura 6.14 se muestra el flujo de datos en un procesador con acumulador en las instrucciones con referencia a memoria.

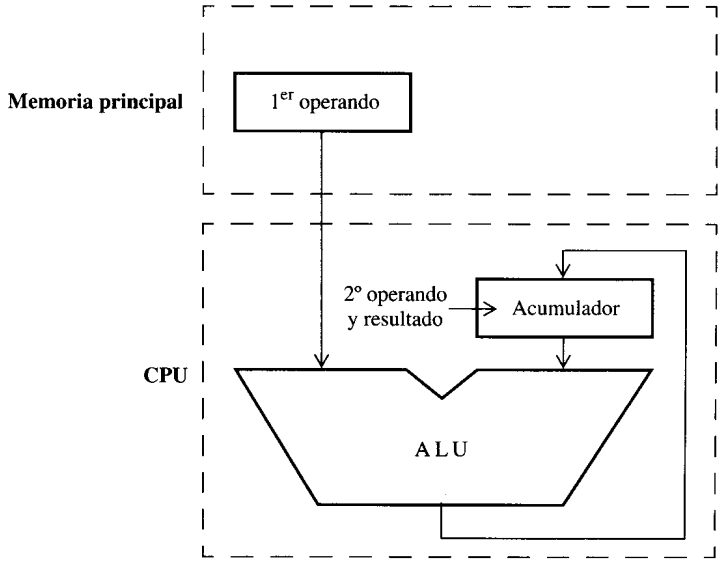


Figura 6.14: Flujo de datos en un procesador de una dirección en las instrucciones con referencia a memoria

6.1.4 Procesadores de cero direcciones (procesadores con pila)

Este tipo de procesador es radicalmente diferente de los analizados en los párrafos precedentes. En efecto no posee ni acumulador ni registros de datos de utilización general. En principio puede parecer imposible considerar un procesador en el que las instrucciones no tengan direcciones explícitas (ver Figura 6.15). Sin

embargo lo único que se requiere, en un procesador de cero direcciones, es saber dónde están los dos operandos y dónde almacenar el resultado.

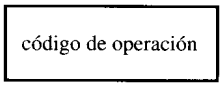


Figura 6.15: Formato de una instrucción de cero direcciones

Los datos se guardan en una *pila* (*stack*) y todas las microoperaciones del procesador se realizan con los datos que están colocados en la *cumbre de la pila* (*SP*) o en el nivel inmediatamente inferior (*SP*-1). El resultado de una microoperación se guarda siempre en la cumbre de la pila. Para comparar los procesadores de pila con los otros procesadores que se han analizado se supondrá que las variables están asociadas con posiciones de la memoria. De esta manera se requieren dos instrucciones primitivas especiales para transferir valores entre la memoria y la pila. Estas instrucciones son las siguientes:

1) *Push M*[*dir*]. Guarda el valor almacenado en la posición de memoria *dir* en la cumbre de la pila:

$$SP \leftarrow M[dir]$$

2) *Pop M*[*dir*]. Saca el valor que está en la cumbre de la pila y lo almacena en la posición de memoria *dir*:

$$M[dir] \leftarrow SP$$

Una característica esencial de los procesadores de pila es que *todas sus operaciones se realizan con los datos que están colocados en la cumbre de la pila* (*SP*) *o en el nivel inmediatamente inferior* (*SP*-1). El resultado de una operación del procesador se guarda siempre en la cumbre de la pila .

$$SP \leftarrow F(SP, SP\text{-}1) \quad \text{ó} \quad SP \leftarrow F(SP)$$

La principal ventaja de este tipo de procesador es que elimina la necesidad de indicar las direcciones de los operandos ya que se encuentran siempre en la cumbre de la pila. Por el contrario la necesidad de trabajar siempre con la cumbre de la pila puede disminuir la velocidad de ejecución de un proceso que demanda varios datos diferentes. En la Figura 6.16, se muestra que el cálculo de $a := a \times b + c \times d \times e$ en un procesador de pila emplea diez instrucciones.

	Contenido de la pila
1. *Push M[A];*	*M[A]*
2. *Push M[B];*	*M[A], M[B]*
3. *Mult;*	*M[A]* × *M[B]*
4. *Push M[C];*	*M[A]* × *M[B], M[C]*
5. *Push M[D];*	*M[A]* × *M[B], M[C], M[D]*
6. *Mult;*	*M[A]* × *M[B], M[C]* × *M[D]*
7. *Push M[E];*	*M[A]* × *M[B], M[C]* × *M[D], M[E]*
8. *Mult;*	*M[A]* × *M[B], M[C]* × *M[D]* × *M[E]*
9. *Add;*	*M[A]* × *M[B]* + *M[C]* × *M[D], M[E]*
10. *Pop M[A];*	

Figura 6.16: Programa que calcula $a := a \times b + c \times d \times e$ en un procesador de pila

Con las mismas suposiciones que en los casos anteriores, cada instrucción sólo necesita ahora una palabra de memoria, lo que significa que el procesador accede diez veces a memoria para traer las instrucciones y otras seis para obtener los operandos y almacenar el resultado (las operaciones *Mult* y *Add* no necesitan realizar ningún acceso a memoria dado que sus operandos se encuentran ya en la pila). En total se precisan pues dieciseis accesos a memoria. Según esto, éste fragmento de programa requiere un tiempo igual que el que emplea el procesador con acumulador y menor que el de los procesadores con dos y tres direcciones.

La pila puede estar realizada en hardware o simulada en una memoria RAM. Por una parte, los procesadores de pila (que incluyen una *pila cableada*, es decir realizada en hardware) ofrecen una alternativa interesante para la evaluación de expresiones aritméticas (ver sección 2-7). Por otra, la simulación de una pila en la memoria principal es el mecanismo más utilizado para resolver el problema de almacenar la dirección de retorno en el salto a subprograma (ver sección 2-7) y en el tratamiento y servicio de interrupciones anidadas (ver apartado 3.4.6).

Como en el caso del procesador con acumulador, tampoco ahora se requiere una variable temporal explícita. Sin embargo a diferencia del acumulador, la pila puede contener cualquier número de valores temporales (únicamente limitado por el tamaño de la pila) y por lo tanto nunca se necesitarán variables temporales explícitas en un procesador de pila. En la Figura 6.17 se muestra el flujo de datos en un procesador con cero direcciones en las instrucciones que efectúa una operación binaria sobre los datos almacenados en la pila (se ha supuesto que la pila está implementada en la memoria principal). Algunas calculadoras programables utilizan un procesador de este tipo para la evaluación de expresiones aritméticas en *notación polaca inversa* o RPN (Reverse Polish Notation).

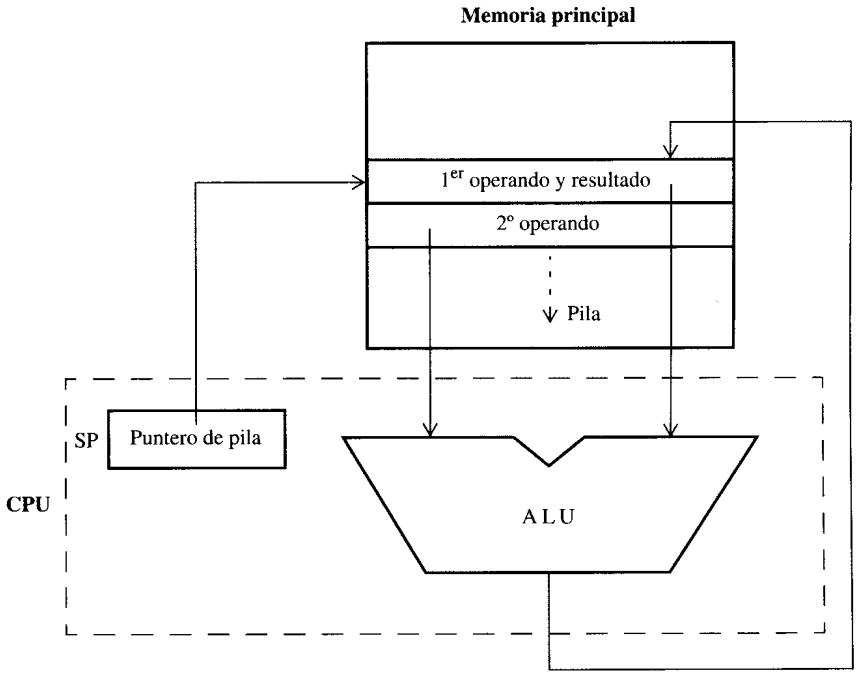


Figura 6.17: Flujo de datos en un procesador de pila en las instrucciones con referencia a memoria

6.1.5 Procesadores sin ALU

Existe una categoría especial de procesadores que no poseen ALU. Las transformaciones realizadas sobre los datos, en estos procesadores, son simples microperaciones de transferencias: el contenido de una posición de memoria se cambia sin que se efectúe una operación lógica o aritmética sobre el valor antiguo. La expresión que define las microoperaciones del procesador en este caso es:

$$R[j] \leftarrow R[i]$$

La simplicidad de la estructura que se deriva de esta definición (ver Figura 6.18) está descompensada por la lentitud del procesador: si son necesarias unas transformaciones aritméticas o lógicas sobre los datos estas deben ser el resultado de una serie de transferencias entre los registros y no el resultado de una simple operación a través de un operador que ha realizado la transformación. A pesar de esta desventaja esta solución se adapta bien a ciertos problemas de control de procesos que demandan pocas operaciones numéricas (o en todo caso poco complejas).

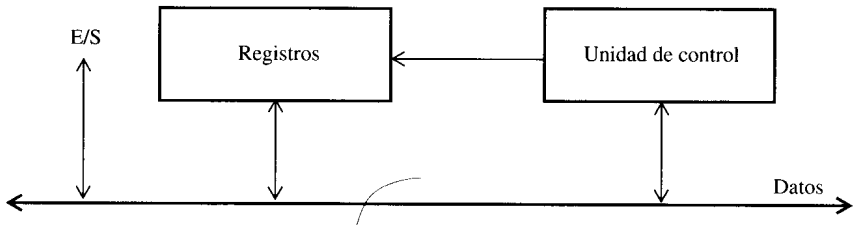


Figura 6.18: Estructura de un procesador sin ALU

6.1.6 Análisis de las diferentes arquitecturas de procesadores

Cada uno de los procesadores que se han presentado en los apartados anteriores tienen instrucciones para evaluar las operaciones binarias usuales y almacenar el resultado. Estas operaciones se denominan ternarias porque es necesario especificar tres operandos: los dos de la operación binaria y un tercero que designa dónde hay que almacenar el resultado. En un procesador de tres direcciones, como ya se ha visto, los tres operandos aparecen de forma explícita en la instrucción. En un procesador de dos direcciones, una dirección especifica al mismo tiempo uno de los operandos fuentes y el operando destino. En los procesadores de una dirección, el acumulador es el segundo operando fuente y también el operando destino. Finalmente en una máquina de cero direcciones, la pila es quien proporciona los dos operandos fuentes y el destino.

En el contexto de los procesadores de tres direcciones no es preciso analizar las operaciones de tipo binario (p. ej. copia, negación, etc). En estos procesadores esta clase de instrucciones se especificarían con sólo dos direcciones. En realidad el que un procesador sea de n direcciones no significa que necesariamente cada una de las instrucciones de su repertorio deban tener n direcciones explícitas. También en el caso de los procesadores de pila es posible que haya alguna instrucción con una dirección explícita (por ejemplo *Push M[dir]* y *Pop M[dir]*) y sin embargo se clasifica el procesador como de cero direcciones. El esquema de clasificación de los procesadores que se ha presentado se basa en cuántas direcciones se necesitan de manera explícita en la mayoría de las instrucciones aritméticas "importantes".

Si se comparan las secuencias de código de las Figuras 6.7, 6.10, 6.13 y 6.16 se observan dos tendencias claras. En primer lugar, el número de instrucciones necesarias parece aumentar conforme disminuye el número de operandos explícitos en cada instrucción. En segundo lugar, el tamaño de cada instrucción parece

disminuir al reducir el número de campos de dirección que posee (ya que no es necesario especificar tantas direcciones). Aunque pueden existir excepciones a estas dos reglas en general suelen ser válidas.

6.1.7 Procesadores con banco de registros

La idea de almacenamiento temporal que subyace en los procesadores con acumulador puede generalizarse incrementando el número de registros al introducir como elemento de la ruta de datos un *banco de registros*. Esto implica que habría que incluir en el formato de las instrucciones campos adicionales para referenciar a estos registros. La utilización de un banco de registros en el diseño del procesador presenta las siguientes ventajas:

1) El acceso a memoria es lento en comparación con la velocidad de la CPU. Para compensar esta diferencia, muchos procesadores cuentan con un pequeño banco de registros dentro de la propia CPU, cada uno con una dirección única de la misma forma que las palabras de la memoria principal. En estos registros se pueden almacenar resultados intermedios y como son parte interna de la CPU para tener acceso a ellos no hace falta el tiempo adicional de un acceso a memoria.

2) Además del ahorro en el tiempo de ejecución de los programas, la utilización de un banco de registros con frecuencia también requiere menos espacio de memoria. Como el número de registros es mucho más pequeño que el número de palabras en la memoria principal, la dirección de un registro es más pequeña que la dirección de una palabra de memoria. Este ahorro se refleja directamente en la codificación de las instrucciones.

Se va a considerar cómo se pueden mejorar los fragmentos de programa de las Figuras 6.7, 6.10, 6.13 y 6.16 cuando se dispone en cada una de las diferentes alternativas de un banco de registros. En este sentido es útil considerar como medida de esta mejora el número de accesos a memoria que son necesarios para ejecutar dichos fragmentos de programa. Cuantos más accesos a memoria se requieran, más tardará en ejecutarse el código. Para simplificar el análisis se supondrá que la codificación de la instrucción en todos los casos ocupa el mismo número de bits y se considera despreciable el tiempo de acceso a los registros del banco de registros que está dentro del propio procesador. De esta forma, la diferencia de comportamiento estará sólo en el número de accesos a memoria que se necesita para localizar a los operandos.

Con el fin de facilitar la comparación se muestra el código del programa en ambas versiones (sin banco de registros y con banco de registros). En las Figuras 6.19, 6.20 y 6.21 se muestran respectivamente las comparaciones cuando se emplean procesadores de tres, dos y una dirección. Se observa que en todos los casos mejora el número de accesos a memoria.

Sin banco de registros	**Con banco de registros**
1. *Mult A, A, B*; $(M[A] \leftarrow M[A] \times M[B])$	1. *Mult R1, A, B*; $(R[1] \leftarrow M[A] \times M[B])$
2. *Mult X, C, D*; $(M[X] \leftarrow M[C] \times M[D])$	2. *Mult R2, C, D*; $(R[2] \leftarrow M[C] \times M[D])$
3. *Mult X, X, E*; $(M[X] \leftarrow M[X] \times M[E])$	3. *Mult R2, E, R2*; $(R[2] \leftarrow M[E] \times R[2])$
4. *Add A, A, X*; $(M[A] \leftarrow M[A] + M[X])$	4. *Add A, R2, R1*; $(M[A] \leftarrow R[2] + R[1])$
12 accesos a memoria	6 accesos a memoria

Figura 6.19: Comparación del cálculo de $a := a \times b + c \times d \times e$ en un procesador de tres direcciones

Sin banco de registros		Con banco de registros	
1. *Mult A, B;*	$(M[A] \leftarrow M[A] \times M[B])$	1. *Mult A, B;*	$(M[A] \leftarrow M[A] \times M[B])$
2. *Move X, C;*	$(M[X] \leftarrow M[C])$	2. *Move R1, C;*	$(R[1] \leftarrow M[C])$
3. *Mult X, D;*	$(M[X] \leftarrow M[X] \times M[D])$	3. *Mult R1, D;*	$(R[1] \leftarrow R[1] \times M[D])$
4. *Mult X, E;*	$(M[X] \leftarrow M[X] \times M[E])$	4. *Mult R1, E;*	$(R[1] \leftarrow R[1] \times M[E])$
5. *Add A, X;*	$(M[A] \leftarrow M[A] + M[X])$	5. *Add A, R1;*	$(M[A] \leftarrow M[A] + R[1])$
	14 accesos a memoria		8 accesos a memoria

Figura 6.20: Comparación del cálculo de $a := a \times b + c \times d \times e$ en un procesador de dos direcciones

Sin banco de registros		Con banco de registros	
1. *Load A;*	$(Acum \leftarrow M[A])$	1. *Load A;*	$(Acum \leftarrow M[A])$
2. *Mult B;*	$(Acum \leftarrow Acum \times M[B])$	2. *Mult B;*	$(Acum \leftarrow Acum \times M[B])$
3. *Store A;*	$(M[A] \leftarrow Acum)$	3. *Store R1;*	$(R[1] \leftarrow Acum)$
4. *Load C;*	$(Acum \leftarrow M[C])$	4. *Load C;*	$(Acum \leftarrow M[C])$
5. *Mult D;*	$(Acum \leftarrow Acum \times M[D])$	5. *Mult D;*	$(Acum \leftarrow Acum \times M[D])$
6. *Mult E;*	$(Acum \leftarrow Acum \times M[E])$	6. *Mult E;*	$(Acum \leftarrow Acum \times M[E])$
7. *Add A;*	$(Acum \leftarrow Acum + M[A])$	7. *Add R1;*	$(Acum \leftarrow Acum + R[1])$
8. *Store A;*	$(M[A] \leftarrow Acum)$	8. *Store A;*	$(M[A] \leftarrow Acum)$
	8 accesos a memoria		6 accesos a memoria

Figura 6.21: Comparación del cálculo de $a := a \times b + c \times d \times e$ en un procesador de una dirección

6.1.8 Arquitectura de carga/almacenamiento: Procesadores RISC

Durante los años 60, 70 y 80 la evolución de las CPU's de los computadores se caracterizó por un incremento notable de su repertorio de instrucciones, tanto en cantidad (varios centenares de instrucciones) como en modos de direccionamiento (ver sección 6-2). Esta tendencia se debió básicamente a razones pragmáticas: cuantas más posibilidades ofrezca el procesador más fácil será construir los niveles superiores de la jerarquía de un sistema de cálculo (compiladores, sistemas operativos, programas de aplicación, etc). No obstante una gran cantidad de estudios empíricos ponían de manifiesto que un gran número de estas instrucciones se utilizaban muy poco.

Esta observación práctica llevó a proponer una alternativa de diseño radicalmente diferente: reducir el repertorio de instrucciones no incluyendo aquellas que tuviesen un porcentaje muy bajo de utilización. Con esta estrategia lo que parece que se consigue es tener programas más largos y en principio más lentos en su ejecución. Sin embargo el diseño del procesador podrá ser mucho más sencillo y eficiente. Como consecuencia de esto el resultado final puede ser que se compensen las desventajas y que el procesador no resulte tan lento. A estos procesadores se les conoce como procesadores RISC[2] que suelen tener del orden de cien instrucciones y un número reducido de modos de direccionamiento, aunque esta característica no es la única ni siquiera la más importante. Otras características de los procesadores RISC son:

2. RISC es el acrónimo de (*Reduced Instruction Set Computer*).

- Tienen una arquitectura de *carga/almacenamiento*. Esto quiere decir que los únicos accesos a la memoria principal son para extraer instrucciones y datos y para almacenar datos. Todas las operaciones de procesamiento se realizan en los registros del procesador.
- Sus instrucciones son *sencillas*. Sólo realizan las operaciones básicas al contrario que los procesadores CISC[3] en los que es norma habitual encontrar instrucciones muy complejas cuyo objetivo es simplificar el proceso de traducción de determinados lenguajes de alto nivel aunque su porcentaje de uso sea muy bajo.
- Su formato de instrucciones es *regular*. Todas las instrucciones tienen la misma longitud y el número de formatos diferentes es reducido (tres o cuatro). En cada formato, todos los bits tienen igual significado para todas las instrucciones y esto implica que la unidad de control sea más sencilla.
- Como consecuencia de lo anterior, y ya al nivel de microprogramación (ver tema 7), la implementación de la unidad de control suele hacerse *cableada* y el número de ciclos por instrucción suele ser de 1.

Si se usa un banco de registros con tres o más registros, se podría calcular $a := a \times b + c \times d \times e$ con la secuencia de instrucciones que se muestra en la Figura 6.22. Una característica notable de este último programa es que sólo necesita instrucciones de *carga* (*Move*) y *almacenamiento* (*Store*) para acceder a memoria. Las instrucciones aritméticas (*Mult* y *Add*) sólo se utilizan con los registros del banco y requieren por lo tanto campos de dirección más cortos. Teniendo esto en cuenta se podrían utilizar instrucciones con tres direcciones para aquellas instrucciones que cargan sus operandos y almacenan sus resultados en registros, e instrucciones con dos direcciones cuando se transfiere un dato desde una posición de memoria a un registro del banco o viceversa. Esta política es la que utilizan la mayoría de los procesadores comerciales que se encuentran actualmente en el mercado ya que se traduce en instrucciones más cortas y en una menor frecuencia de accesos a memoria.

1.	*Move R1, A;*	$(R[1] \leftarrow M[A])$
2.	*Move R2, B;*	$(R[2] \leftarrow M[B])$
3.	*Move R3, C;*	$(R[3] \leftarrow M[C])$
4.	*Move R4, D;*	$(R[4] \leftarrow M[D])$
5.	*Move R5, E;*	$(R[5] \leftarrow M[E])$
6.	*Mult R1, R1, R2;*	$(R[1] \leftarrow R[1] \times R[2])$
7.	*Mult R3, R3, R4;*	$(R[3] \leftarrow R[3] \times R[4])$
8.	*Mult R3, R3, R5;*	$(R[3] \leftarrow R[3] \times R[5])$
9.	*Add R1, R1, R3;*	$(R[1] \leftarrow R[1] + R[3])$
10.	*Store A, R1;*	$(M[A] \leftarrow R[1])$
		6 accesos a memoria

Figura 6.22: Programa que calcula $a := a \times b + c \times d \times e$ en una arquitectura de carga/almacenamiento

En la Tabla 6.1 se resumen de forma comparativa las propiedades que caracterizan al repertorio de instrucciones de un procesador RISC frente a un procesador CISC.

3. CISC es el acrónimo de (*Complex Instruction Set Computer*).

Procesador RISC	Procesador CISC
1) Los accesos a memoria se restringen a instrucciones de carga/almacenamiento y las de manipulación de datos son entre registros.	1) La mayoría de los tipos de instrucciones permiten que el acceso a memoria sea de forma directa.
2) Existe un número limitado de modos de direccionamiento.	2) Hay un número considerable de modos de direccionamiento.
3) Los formatos de las instrucciones tienen todos la misma longitud.	3) Los formatos de las instrucciones tienen longitudes diferentes.
4) Las instrucciones realizan operaciones elementales.	4) Las instrucciones realizan operaciones elementales y complejas.

Tabla 6.1: Comparación de las características de un procesador RISC frente a un procesador CISC

En el ejemplo que se ha venido mostrando a lo largo de toda esta sección, se han propuesto diversas soluciones para tratar de satisfacer objetivos que entran en conflicto y que aparecen al intentar acelerar la ejecución de los programas utilizando el máximo número posible de direcciones por instrucción. Como pone de relieve esta última opción presentada, se llega a un buen compromiso si se emplea un pequeño banco de registros como almacenamiento temporal que permite emplear instrucciones con tres direcciones para las operaciones sobre los datos guardados en estos registros (porque las direcciones del banco de registros necesitan menos bits) y usar instrucciones de carga y almacenamiento con una dirección de memoria para transferir datos entre el banco de registros y la memoria principal.

Esta estrategia de separar las instrucciones del procesador en instrucciones de carga/almacenamiento e instrucciones con registros se fundamenta en la observación de que cada variable del programa se utiliza más de una vez y el acceso a dichas variables en un banco de registros de acceso rápido con direcciones cortas disminuye los accesos a memoria principal.

Sección 6-2

Modos de direccionamiento

Tal como se ha visto ya en el formato de cualquier instrucción el *código_op* especifica la operación que se va a realizar y el *campo de dirección* proporciona la información precisa para conocer la localización de los operandos y del resultado de la operación. La forma en que se interpreta el contenido del *campo de dirección* viene determinada por el *modo de direccionamiento* que permite calcular de forma no ambigua la *dirección real* o *efectiva* donde se encuentran los operandos de la instrucción y dónde hay que almacenar el resultado que produce. En los computadores actuales existe una gran variedad de modos de direccionamiento que tienen como finalidad optimizar el rendimiento global de la máquina. Entre las ventajas más sobresalientes se pueden mencionar las siguientes:

a) *Reducción del tamaño de palabra de la instrucción.* El campo de modo de direccionamiento permite reducir el tamaño del campo de dirección de manera que sólo se requiere explicitar en el formato de la instrucción una parte de la dirección. El modo de direccionamiento da un método para calcular la dirección real a partir de la información contenida en el campo de dirección.

b) *Aumento de la flexibilidad en la programación.* Con la introducción de los diferentes modos de direccionamiento el usuario tiene más grados de libertad para escribir programas eficientes y sencillos de modificar. Los modos de direccionamiento entre otras cosas permiten disponer fácilmente de punteros a memoria, contadores para controlar la ejecución de un bucle, indexación de datos y reubicación de programas en distintas zonas de memoria.

En algunos computadores además del campo *código_op* (que especifica la operación que se realiza), se emplea un campo de *modo de direccionamiento* (que determina la dirección efectiva de los operandos que se necesitan para la operación). Otra alternativa es emplear sólo el campo *código_op* lo que implica que la distinción entre los modos de direccionamiento se hace empleando diferentes codificaciones para dicho campo. La instrucción puede o no contener un campo de dirección. Si hay un campo de dirección, puede designar a una dirección de memoria o a un registro del procesador. Más aún, tal como se analizó en la sección anterior, la instrucción puede tener más de un campo de dirección. En ese caso, cada campo de dirección se asocia con su propio modo particular de direccionamiento.

En la Figura 6.23 se muestra como ejemplo el formato de las instrucciones de un procesador genérico de 16 bits. Cada instrucción consta de una o dos palabras de 16 bits; la segunda palabra, en caso de que se utilice, sería una dirección de memoria, mientras que la primera palabra especifica los elementos siguientes: tipo de instrucción, código de operación de la instrucción (que identifica implícitamente el modo de direccionamiento) y tres direcciones de un banco de registros. Para acomodar las tres direcciones del banco de registros, hay que dividir la instrucción en cinco campos: el campo de *tipo* (2 bits), el campo de *código_op* (5 bits) y tres direcciones del banco de registros identificadas como *destino* (3 bits), *fuente1* (3 bits) y *fuente2* (3 bits).

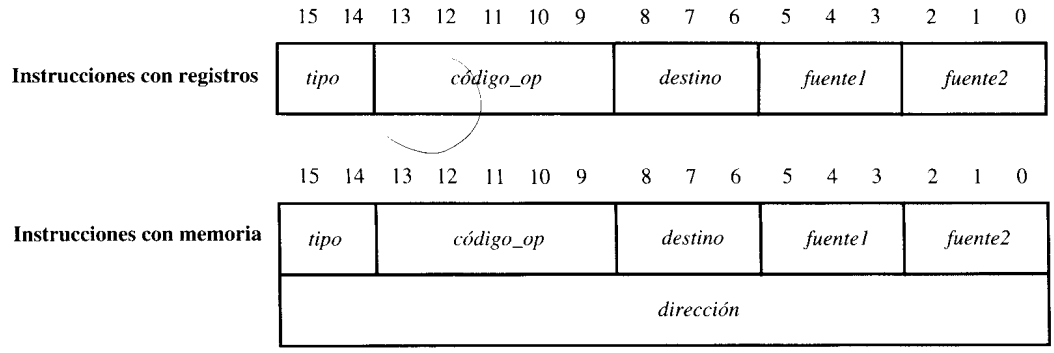


Figura 6.23: Ejemplo de formatos de instrucciones de un procesador de 16 bits

A continuación se describen de forma sucinta los diferentes modos de direccionamiento.

1) *Direccionamiento implícito.* Aunque la mayoría de los modos de direccionamiento modifican el campo de dirección de la instrucción, hay un modo que no necesita en absoluto de este campo: el modo implícito. En este modo el propio *código_op* especifica de forma implícita la posición del operando o del resultado (ver Figura 6.24). Por ejemplo, una instrucción que opera sobre el acumulador no necesita un campo de dirección ya que este registro es único y por lo tanto está designado implícitamente en el propio *código_op*. Este tipo de modo de direccionamiento es característico de los procesadores de pila en los que todos los operandos deben estar en la cumbre de la pila y el resultado se almacena también de forma automática en la cumbre de la pila.

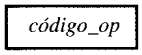


Figura 6.24: Direccionamiento implícito

2) *Direccionamiento inmediato.* En el modo de direccionamiento inmediato el campo de dirección lo que realmente contiene es el propio operando en lugar de una dirección o cualquier otra información que describa dónde se encuentra el operando (ver Figura 6.25). El direccionamiento inmediato es la forma más simple que tiene una instrucción de especificar un operando. Este tipo de direccionamiento resulta interesante cuando se incrementan o decrementan índices de un bucle o de un vector o cuando se calculan expresiones que tienen muchos coeficientes ya que los índices y los coeficientes se pueden dar directamente en el propio campo de dirección en lugar de estar almacenados en la memoria. Este esquema de direccionamiento ahorra así accesos innecesarios a memoria por parte del procesador.

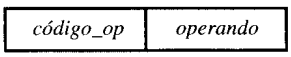


Figura 6.25: Direccionamiento inmediato

3) *Direccionamiento directo.* También se le conoce como *direccionamiento absoluto.* En este modo de direccionamiento el campo de dirección contiene la dirección de la posición de memoria o la dirección del registro donde se almacena el operando o el resultado (ver Figura 6.26). Cuando el campo de dirección se refiere a un registro (del banco de registros), este modo se denomina *direccionamiento directo a través de registro.* Conviene recordar que una dirección de memoria es mucho más larga que una dirección de registro, ya que si los tamaños de la memoria varían entre $16M$ y $512M$ palabras entonces los tamaños de los bancos de registros van desde 8 a 256 palabras. Esto quiere decir que se necesitan de 24 a 28 bits para una dirección de memoria y sólo de 3 a 8 bits para una dirección de registro. Este tipo de direccionamiento sólo requiere de una referencia a memoria (o registro) para buscar el operando y no necesita hacer ningún cálculo especial para obtener la dirección del operando. Sin embargo proporciona un espacio de direccionamiento limitado que depende del tamaño del campo de dirección (si el campo de dirección posee n bits se pueden direccionar, como máximo, de forma directa 2^n palabras de memoria).

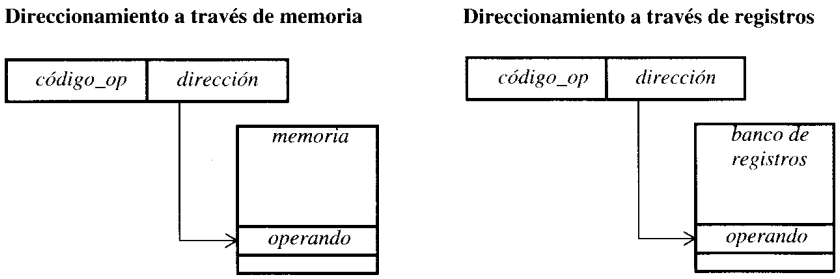


Figura 6.26: Direccionamiento directo

4) *Direccionamiento relativo.* En el direccionamiento relativo el contenido del campo de dirección que en este caso se denomina *desplazamiento* se suma al contenido de un registro que se especifica en la instrucción como por ejemplo el contador de programas o un registro del banco de registros (ver Figura 6.27). Este desplazamiento es normalmente un número entero pequeño que cuando se suma al contenido del contador de programas genera la dirección de una instrucción que está cerca de la instrucción original. Por tanto, este tipo de direccionamiento se emplea en las instrucciones de transferencia de control ya que frecuentemente la dirección a la que se produce el salto está en las cercanías de la propia instrucción de bifurcación. El direccionamiento relativo se puede utilizar también con respecto a un registro del banco de registros. En este caso se utiliza para realizar búsquedas en tablas; normalmente el registro contiene el inicio de la tabla y el desplazamiento se emplea para apuntar a un elemento concreto de la misma. Otra aplicación característica del direccionamiento relativo es en la reubicación de código de forma que el registro contiene la primera instrucción y el desplazamiento se usa para apuntar a la siguiente instrucción que hay que ejecutar. La reubicación de código permite mover programas dentro de la memoria del computador, conmutar entre diferentes programas y alternar su ejecución. La ventaja que presenta este modo de direccionamiento es que da lugar a un formato de instrucción que tiene un campo de dirección más pequeño ya que el desplazamiento necesita menos bits que la dirección de memoria completa.

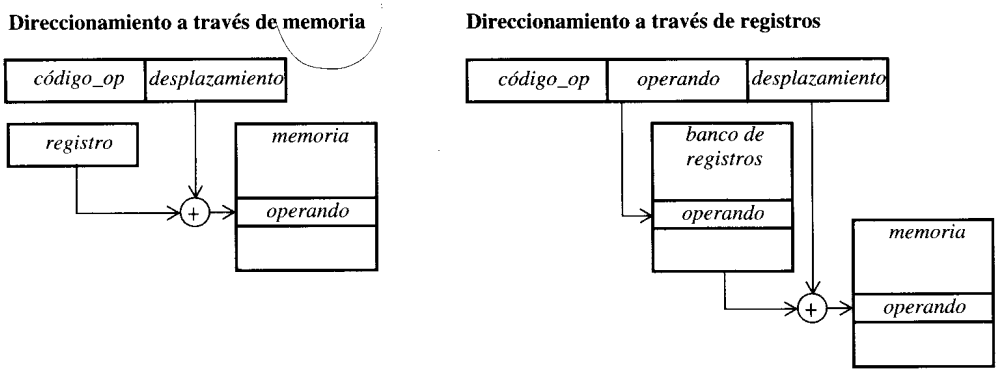


Figura 6.27: Direccionamiento relativo

5) *Direccionamiento indirecto.* En el direccionamiento indirecto el campo de dirección especifica que posición de memoria o registro contiene la dirección donde hay que ir a buscar el operando o donde hay que almacenar el resultado (ver Figura 6.28). Cuando el campo de dirección se refiere a un registro (del banco de registros), este modo se denomina *direccionamiento indirecto a través de registro.* La principal ventaja de utilizar el direccionamiento indirecto a través de registro es que la dirección del registro necesita menos bits que una dirección de memoria y que el acceso al banco de registros es mucho más rápido. En este modo de direccionamiento, el espacio de direccionamiento del operando depende de la longitud de palabra de la memoria o de la longitud del registro. Al contrario que con el direccionamiento directo, con este esquema el espacio de direccionamiento no está limitado por el tamaño del campo de dirección. Su principal desventaja es que se necesitan dos referencias para localizar al operando, una referencia a memoria (o registro) para conseguir la dirección del operando y una segunda referencia a memoria para obtener el propio operando.

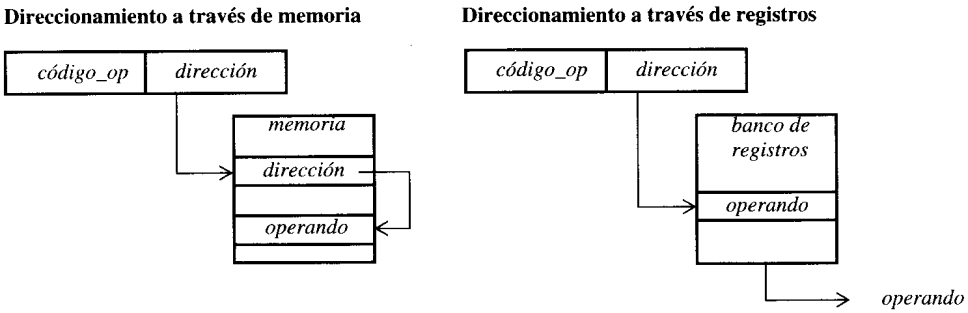


Figura 6.28: Direccionamiento indirecto

6) *Direccionamiento indexado.* El direccionamiento indexado se emplea normalmente cuando se precisa acceder a datos que están almacenados en arrays, vectores, pilas o colas. El campo de dirección de la instrucción especifica la dirección de comienzo denominada *base*, mientras que el

índice del dato se encuentra en un registro índice específico o mediante un registro del banco de registros. El cálculo de la dirección efectiva se efectúa sumando el contenido del registro índice al valor de la base (ver Figura 6.29). Si el contenido del registro se incrementa o decrementa de forma automática después de la ejecución de la instrucción se dice que el direccionamiento indexado posee *autoincremento* o *autodecremento*. El direccionamiento indexado también disminuye el número de bits que son necesarios en el campo de dirección, ya que las direcciones de las bases suelen ser múltiplos de 2^n, conteniendo $(n\text{-}1)$ ceros. Como estos $(n\text{-}1)$ ceros no necesitan almacenarse en el campo de dirección la instrucción requiere un número menor de bits. Este tipo de direccionamiento es similar al direccionamiento relativo ya que en ambos casos los contenidos del campo de dirección y de un determinado registro tienen que sumarse para generar la dirección efectiva. La diferencia está en la situación de la base y del índice o desplazamiento. En el direccionamiento relativo, la base está en el registro y el desplazamiento se coloca en el campo de dirección mientras que en el direccionamiento indexado, la base está en el campo de dirección y el índice en un registro.

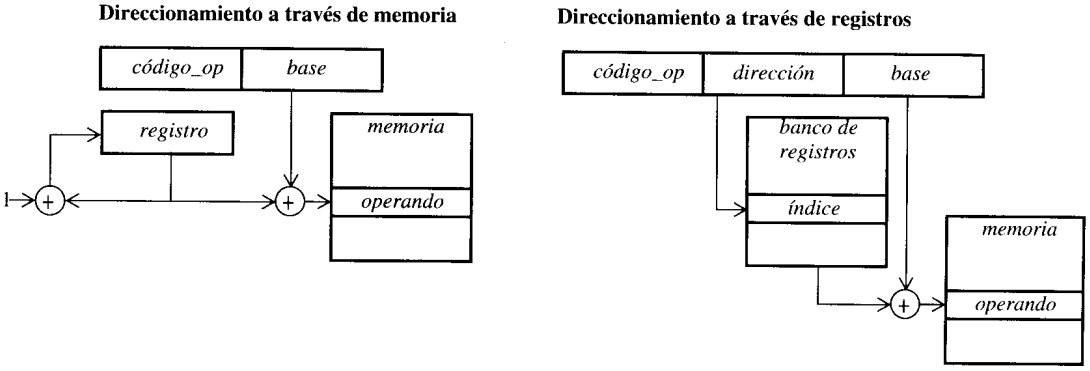


Figura 6.29: Direccionamiento indexado

Sección 6-3

Ciclo de ejecución de una instrucción

La ejecución de una instrucción siempre conlleva realizar la misma secuencia de pasos independientemente del repertorio de instrucciones específicos que posea el procesador y de los campos y modos de direccionamientos que se hayan definido. De forma genérica, el *ciclo de ejecución de una instrucción* se puede subdividir en cuatro fases o pasos principales. En cada uno de estos pasos se lleva a cabo una serie de operaciones en los distintos elementos que constituyen la CPU que dan como resultado final la ejecución de la instrucción. Cada uno de estos pasos puede necesitar un número diferente de ciclos de reloj dependiendo de su propia complejidad y de los recursos que la CPU tenga para su realización. La temporización básica del computador viene dada por un *reloj*, que determina el tiempo mínimo que puede durar una operación elemental. Cada uno de estos tiempos elementales se denomina *periodo*. En la Figura 6.30 se muestran las fases en que se divide la ejecución de una instrucción.

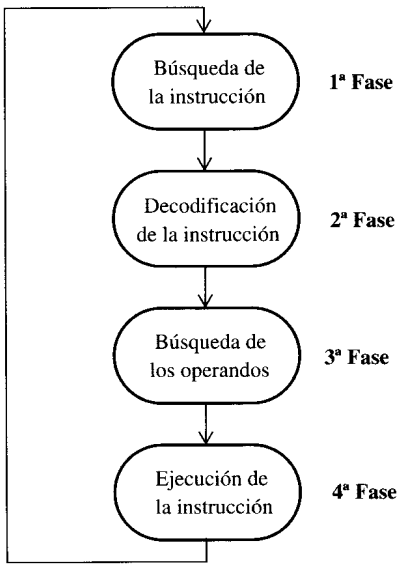


Figura 6.30: Ciclo de ejecución de cualquier instrucción

6.3.1 Fase de búsqueda de la instrucción

En esta fase se realiza la búsqueda en memoria de la instrucción cuya dirección guarda el *registro contador de programa* (*PC*) y se almacena en el *registro de instrucción* (*IR*). El contenido del *PC* se incrementa de

manera que almacena la dirección de la siguiente instrucción en secuencia. El objetivo de esta fase es realizar en el orden indicado las siguientes microoperaciones:

$$1.\ IR \leftarrow M[PC]$$
$$2.\ PC \leftarrow PC + 1$$

En primer lugar se accede a la memoria para leer la posición de memoria a la que apunta el contenido del *PC* y la instrucción contenida en dicha posición se almacena en el registro *IR*. De forma simultánea se incrementa en uno el contenido del registro PC para que contenga la dirección de memoria de la siguiente instrucción que se tiene que ejecutar.

La primera de las dos microperaciones anteriores presupone que necesariamente el contador de programa está conectado al bus de direcciones y que puede acceder a memoria. Sin embargo hay determinadas arquitecturas de computadores que no ofrecen esta posibilidad ya que el acceso a memoria se realiza sólo mediante el *registro de dirección de memoria* (*MAR*) que especifica la próxima dirección de memoria de donde se va a leer o donde se va a escribir. También es preciso utilizar el *registro de datos de memoria* (*MBR*) que está conectado con el bus de datos del computador y contiene el dato a escribir en la memoria o recibe el dato leído de la memoria (en los ejemplos que siguen se supone que todo el intercambio de información con la memoria se hace a traves de MBR). En este caso el conjunto de microoperaciones que define la fase de búsqueda son las siguientes:

$1.\ MAR \leftarrow PC$	Copia *PC* en *MAR*
$2.\ MBR \leftarrow M[MAR]$	Lee la instrucción en *MBR*
$3.\ PC \leftarrow PC + 1$	Incrementa *PC*
$4.\ IR \leftarrow MBR$	Mueve la instrucción a *IR*

En la Figura 6.31 se muestra, para esta última situación, el flujo de datos que se produce en la fase de búsqueda. Los números encerrados en pequeños círculos en la Figura 6.31 indican la transferencia o acción que se produce en la microoperación correspondiente.

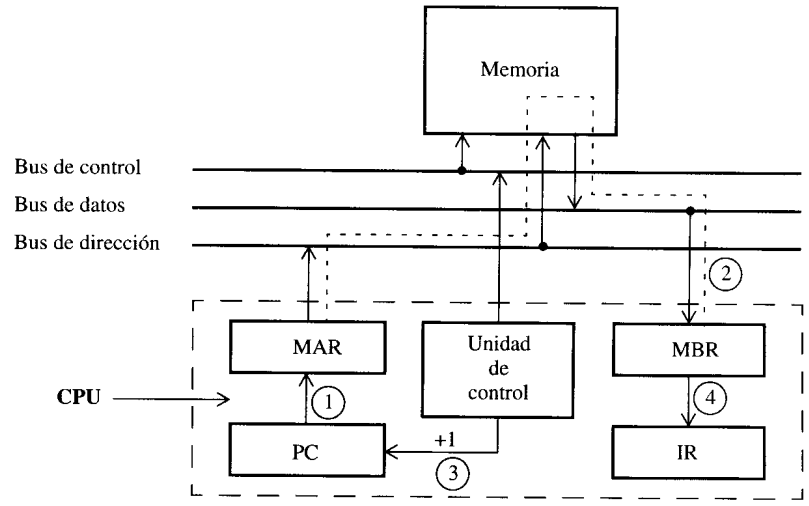


Figura 6.31: Flujo de datos en la fase de búsqueda

En la Figura 6.32 se muestra un ejemplo del efecto que las cuatro microoperaciones de la fase de búsqueda de una instrucción tienen sobre los cuatro registros de la CPU que se acaban de describir.

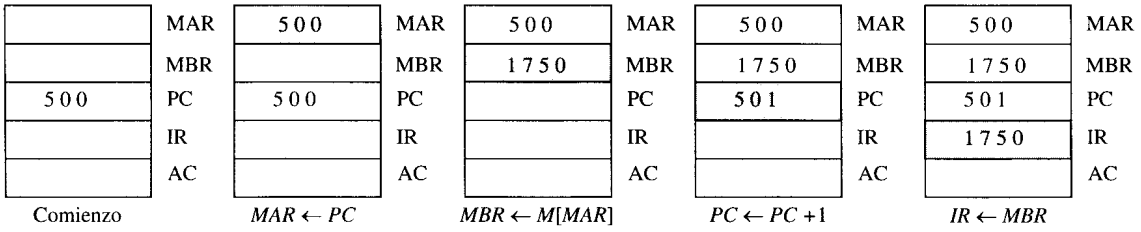


Figura 6.32: Ejemplo del contenido de los registros de la CPU durante la fase de búsqueda

La secuencia de acciones que se produce durante la fase de búsqueda, en el caso de que el contador de programa no esté conectado al bus de direcciones, es la siguiente: Al comienzo el PC contiene la dirección de la próxima instrucción a ejecutar; en el ejemplo la dirección es 500 (se supone que el contenido de esa posición de memoria es 1750). La primera microoperación transfiere esa dirección a MAR (registro de dirección de memoria), puesto que es el único registro que está conectado a las líneas de dirección del bus del sistema. En la segunda microoperación la dirección deseada (que está en MAR) se coloca sobre el bus de direcciones, la unidad de control emite una orden de lectura a través del bus de control, y el resultado aparece en el bus de datos y se carga en MBR (registro de datos de memoria). La tercera microoperación incrementa en 1 el PC y se tiene así ya preparado para la búsqueda de la siguiente instrucción. La cuarta microoperación finalmente transfiere el contenido de MBR a IR.

Cada microoperación implica el movimiento de algún dato desde o hacia alguno de los cuatro registros. Si estos movimientos no interfieren unos con otros, pueden tener lugar dentro de un mismo paso ó ciclo de reloj, lo que acelera la ejecución de la fase de búsqueda. En concreto las microoperaciones segunda y tercera representan dos acciones (lectura de una palabra de la memoria y sumar 1 al PC) que no interfieren entre sí y para ahorrar tiempo se pueden realizar simultáneamente en un mismo paso (ver Tabla 6.2).

Fase de búsqueda	Microoperaciones
Paso 1	MAR ← PC
Paso 2	MBR ← M[MAR] PC ← PC + 1
Paso 3	IR ← MBR

Tabla 6.2: Secuencia de pasos en la fase de búsqueda

Otra alternativa posible es agrupar la tercera y cuarta microoperación (en lugar de la segunda y la tercera) sin afectar por ello a la fase de búsqueda (ver Tabla 6.3). Para agrupar microoperaciones se deben seguir dos reglas simples:

1) *Respetar la secuencia de acciones.* Por ejemplo, *MAR* ← *PC* debe preceder a *MBR* ← *M*[*MAR*], puesto que la operación de lectura de memoria utiliza la dirección almacenada en MAR.

2) *Evitar conflictos.* No se debe intentar leer y escribir simultáneamente en el mismo registro, ya que

los resultados serían impredecibles. Así, las microoperaciones $MBR \leftarrow M[MAR]$ y $IR \leftarrow MBR$ no deben realizarse en el mismo intervalo de tiempo.

Fase de búsqueda	Microoperaciones
Paso 1	MAR ← PC
Paso 2	MBR ← M[MAR]
Paso 3	PC ← PC + 1 IR ← MBR

Tabla 6.3: Secuencia de pasos alternativa en la fase de búsqueda

6.3.2 Fase de decodificación de la instrucción

Las operaciones a realizar en la unidad de procesamiento para ejecutar las instrucciones depende del contenido del campo que identifica el tipo de instrucción. En esta fase la unidad de control analiza el campo código de operación (*código_op*) de la instrucción y determina cuáles son las operaciones que hay que realizar para su correcta ejecución. Como ya se ha dicho el *código_op* puede también contener información adicional sobre la forma de construir la dirección efectiva donde hay que localizar los operandos.

6.3.3 Fase de búsqueda de los operandos

En esta fase, si la instrucción lo precisa, se obtienen los operandos de los registros internos de la CPU, de la memoria o incluso directamente de la propia instrucción si alguno de ellos posee un modo de direccionamiento inmediato. En la Figura 6.33 se muestra el flujo de datos que hay durante esta fase si por ejemplo en la instrucción se especifica un modo de direccionamiento indirecto.

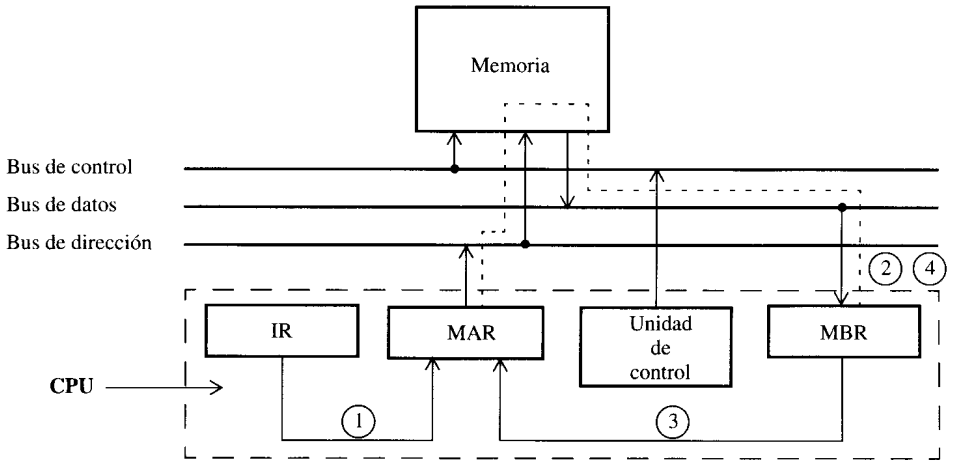


Figura 6.33: Flujo de datos en la fase de búsqueda de los operandos con un modo de direccionamiento indirecto

En la Tabla 6.4 se da la secuencia de pasos que constituye la fase de búsqueda del operando cuando se emplea un modo de direccionamiento indirecto.

Fase de búsqueda del operando	Microoperaciones
Paso 1	MAR ← IR[Dirección]
Paso 2	MBR ← M[MAR]
Paso 3	MAR ← MBR
Paso 4	MBR ← M[MAR]

Tabla 6.4: Secuencia de pasos en la fase de búsqueda de los operandos con direccionamiento indirecto

1) El campo de dirección de la instrucción que está en IR se transfiere a MAR.
2) El contenido de la posición de memoria a la que apunta MAR (que es realmente la dirección donde se encuentra el operando) se transfiere a MBR.
3) Se carga MAR con el contenido de MBR, de forma que ahora apunta a la dirección donde se localiza el operando.
4) Se lee el operando en MBR con lo que finaliza la fase.

Si se hubiese empleado un modo de direccionamiento directo sólo habrían sido necesario los dos primeros pasos. Evidentemente las microoperaciones que son necesarias, en un procesador en concreto, es función de los caminos de transferencias de datos que se establezcan entre sus distintos recursos. En este ejemplo académico se ha supuesto una arquitectura muy simple en el que toda la comunicación con la memoria se realiza únicamente a través de dos registros: MAR (para direccionar memoria) y MBR (para leer o escribir en memoria).

6.3.4 Fase de ejecución de la instrucción

En esta fase se realiza la operación indicada por la instrucción y si ésta lo precisa se almacena el resultado en los registros internos de la CPU o en la memoria.

Las fases de búsqueda de la instrucción, decodificación de la instrucción y búsqueda de los operandos son simples y predecibles. Para un procesador dado cada una de ellas contiene una secuencia fija de acciones y en cada caso, las mismas microoperaciones se repiten una y otra vez. Esto no sucede sin embargo con la fase de ejecución de la instrucción. Para una máquina que posea N códigos de operación diferentes, pueden ocurrir N secuencias distintas de microoperaciones. A continuación se consideran algunos ejemplos ilustrativos. En un procesador con acumulador, sea la instrucción:

$$\text{AND } X \qquad (AC \leftarrow AC \text{ AND } X)$$

que efectúa el AND lógico (bit a bit) del contenido de la posición X con el registro acumulador AC de la máquina y deja el resultado de dicha operación otra vez en el acumulador. Se supone que en la fase de búsqueda de los operandos de esta instrucción se ha dejado ya almacenado el contenido de la posición X en MBR. En la Tabla 6.5 se muestra la fase de ejecución de esta instrucción.

Fase de ejecución	Microoperaciones
Paso 1	AC ← AC AND MBR

Tabla 6.5: Fase de ejecución de la operación AND en un procesador con acumulador

Las *instrucciones de salto* merecen algunas consideraciones particulares. Garantizan el correcto secuenciamiento de las instrucciones de un programa ya que van siendo seleccionadas para su ejecución en un orden determinado. Si el *código_op* especifica una instrucción de bifurcación, la unidad de control decide en esta fase si hay que efectuar o no la ruptura de secuencia en la ejecución del programa en función del tipo de instrucción y del valor del registro de estado. La selección de un camino entre varios posibles y la realización de bucles en un algoritmo, requieren instrucciones capaces de transferir el control a una instrucción distinta de la que le sigue en secuencia. Son las denominadas *instrucciones de salto*, que pueden ser *condicionales* e *incondicionales*. En la ruptura de secuencia incondicional el PC se actualiza con la dirección donde continúa el programa:

$$PC \leftarrow x$$

En la ruptura de secuencia condicional el PC se actualiza con:

$$PC \leftarrow x \qquad \text{si se satisface la condición de salto}$$

donde x es la dirección efectiva a la que hay que bifurcar.

6.3.5 Transferencia a un subprograma

Las rupturas de secuencia condicional e incondicional son válidas para controlar el flujo dentro de un programa. Sin embargo, con frecuencia es necesario realizar un mecanismo de transferencia de control temporal desde un programa principal P a un subprograma S. Existen dos situaciones en las que esto ocurre: los *saltos a subprogramas* y las *interrupciones*. Una llamada a un subprograma (*CALL S*) es una transferencia temporal de control desde P a S, iniciada a petición de P y una interrupción es una transferencia de control temporal desde P a S, iniciada a petición de S o de algún dispositivo asociado con el programa S. Aunque las interrupciones ya han sido suficientemente tratadas en la sección 3-4, en el siguiente apartado se verá cómo se encadena el ciclo de interrupción con el ciclo de ejecución de una instrucción. En cualquier caso, cuando el programa S finaliza su ejecución debe proseguir la ejecución del programa P en el punto en que había quedado suspendido. Por esta razón cuando se ejecuta una instrucción del tipo de salto a subprograma, *CALL S*, las dos operaciones que hay que realizar son:

1) Guardar el contenido actual del PC, que es la dirección de la siguiente instrucción de P, en algún registro de la CPU o en alguna posición predeterminada de la memoria principal (a esta dirección se la denomina *dirección de retorno*).

2) Cargar S en el PC (*PC* ← *S*) lo que origina que se comience la ejecución del subprograma. La instrucción final en el subprograma S debe ser un retorno (*RET*) al programa P que lo llamó. La instrucción *RET* toma la dirección que fue almacenada por la instrucción *CALL S* y la coloca en el PC. Esto resulta en una transferencia otra vez de la ejecución al punto de continuación en el programa P.

Tal como se explicó en el apartado 2.7.1 al hablar de las aplicaciones de las memorias tipo pila, el método más utilizado en la actualidad para realizar la transferencia de control de un programa *P* a otro programa *S* consiste en la utilización de una pila en la que se van almacenando las direcciones de retorno (ver Figura 2.62). La utilización de una pila implica que el procesador almacena de forma automática todas las direcciones de retorno sin que el programador tenga que preocuparse para nada de gestionarlas. Recuérdese que en una pila, la información se recupera en orden inverso a como se introdujo. Esta es precisamente la situación cuando se producen llamadas a subprogramas que están anidados. Siempre hay que volver al último punto de retorno que se introdujo en la pila. En la Figura 6.34 se muestran las secuencias de microoperaciones que tienen lugar en la ejecución de las instrucciones *CALL S* y *RET* cuando se utiliza una pila.

CALL S		*RET*	
$SP \leftarrow SP - 1$	Decrementa el puntero de pila	$PC \leftarrow M[SP]$	Transfiere dirección de retorno a PC
$M[SP] \leftarrow PC$	Almacena la dirección de retorno en la pila	$SP \leftarrow SP + 1$	Incrementa el puntero de pila
$PC \leftarrow S$	Transfiere el control al subprograma S		

Figura 6.34: Microoperaciones de CALL *S* y RET implementadas con una pila

En la Figura 6.35 se muestra una forma alternativa de realizar el retorno desde *S* a *P* cuando el procesador no dispone de una pila. Cuando se ejecuta un salto a un subprograma, *CALL S*, la dirección de retorno (*m*+1) se almacena en la primera palabra del subprograma *S* (*x*). El retorno desde *S* a *P* se produce ejecutando un salto incondicional con direccionamiento indirecto sobre el contenido de la posición *x* (*salto_i x*). Este esquema se utilizó en la década de los 70 en algunos miniordenadores muy populares como el PDP-8 de *DEC* y la serie HP1000 de *Hewlett Packard*, sin embargo presenta un gran inconveniente y es que no permite la ejecución de programas recursivos (un subprograma no puede llamarse a si mismo).

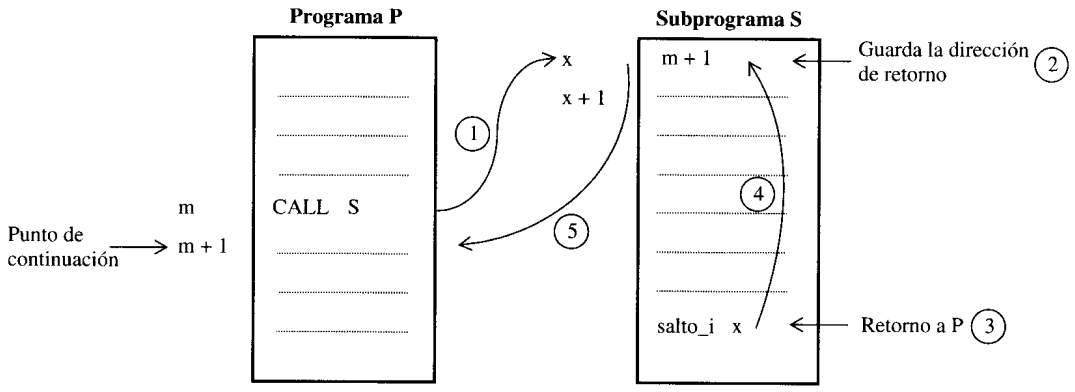


Figura 6.35: Retorno de *S* a *P* mediante un salto incondicional con direccionamiento indirecto

6.3.6 Ciclo de interrupción

La petición de una interrupción externa al procesador (*PI* = 1) se puede producir en cualquier instante de tiempo durante la ejecución de un programa. Para asegurar que no se pierde ninguna información, el

computador normalmente reconoce la interrupción (*RI* = 1) sólo después de que se finaliza la ejecución de la instrucción actual y si el estado del procesador así lo autoriza porque el elemento de memoria *EI* está a 1. *EI* (*Enable Interrupt*) normalmente se incluye dentro de los bits del registro de estado *RE* del procesador y su valor puede ser puesto a 1 (mecanismo de interrupción activado) o a 0 (mecanismo de interrupción desactivado) por instrucciones del programa.

En la Figura 6.36 se muestra un esquema simplificado que sirve para explicar cómo el procesador gestiona el ciclo de interrupción. Los pasos que se siguen son los siguientes:

1) Al procesador llega una señal externa de petición de interrupción (*PI* = 1).
2) Si *EI* = 0 la CPU no considera la petición de interrupción y el ciclo de interrupción finaliza. Si *EI* = 1 y la CPU está al final del ciclo de instrucción, acepta la interrupción poniendo a 1 la línea de reconocimiento de interrupción (*RI* = 1).
3) La fuente de la interrupción (normalmente un controlador de interrupciones) responde a *RI* = 1 proporcionando al procesador una dirección de interrupción *dint*.

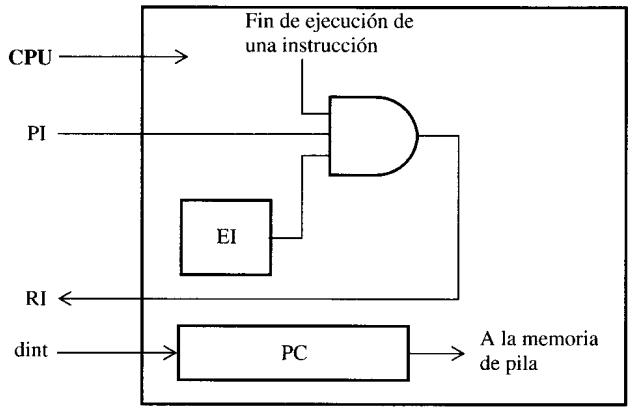


Figura 6.36: Gestión del ciclo de interrupción por el procesador

En la Figura 6.37 se muestran las microperaciones que implementan el ciclo de interrupción cuando se utiliza una pila.

$SP \leftarrow SP - 1$	Decrementa el puntero de pila
$M[SP] \leftarrow PC$	Almacena la dirección de retorno en la pila
$SP \leftarrow SP - 1$	Decrementa el puntero de pila
$M[SP] \leftarrow RE$	Almacena el registro de estado del procesador en la pila
$EI \leftarrow 0$	Mecanismo de interrupción desactivado
$RI \leftarrow 1$	Reconocimiento de interrupción activado
$PC \leftarrow dint$	Transfiere la dirección de interrupción a PC

Figura 6.37: Microoperaciones del ciclo de interrupción

En el organigrama de la Figura 6.38 se resume, de acuerdo con lo visto en esta sección, la conducta del procesador en un ciclo de instrucción y si se precisa de un ciclo de interrupción.

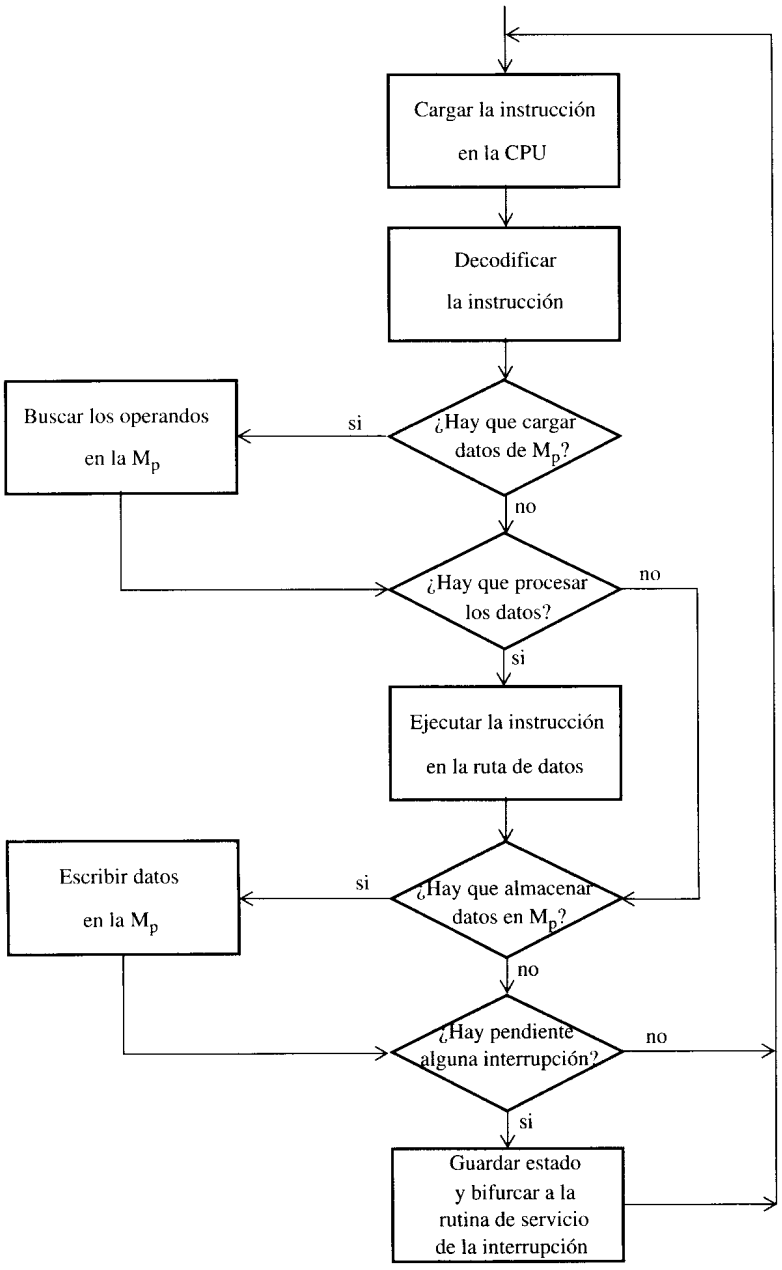


Figura 6.38: Organigrama del secuenciamiento de ciclos de instrucción y/o ciclos de interrupción

Obviamente, habrá que escribir un programa de servicio para el tratamiento de la interrupción. El retorno de la interrupción se hace con una instrucción al final del programa de servicio (*IRET*) que es similar al retorno desde un subprograma (*RET*) que se acaba de considerar anteriormente. La información que hay en la pila se extrae y la dirección de retorno se transfiere al *PC*. Como *EI* está incluido en el registro de estado *RE*, retorna al valor que tenía antes de que se produjese la interrupción, cuando *RE* se restaura desde la pila. De esta forma el sistema de interrupciones estará activado o desactivado para el programa original de la misma forma que lo estaba antes de que ocurriese la interrupción.

Sección 6-4

Fases en el diseño del procesador

El diseño de un procesador comporta una serie de pasos que se muestran de forma resumida en la Figura 6.39.

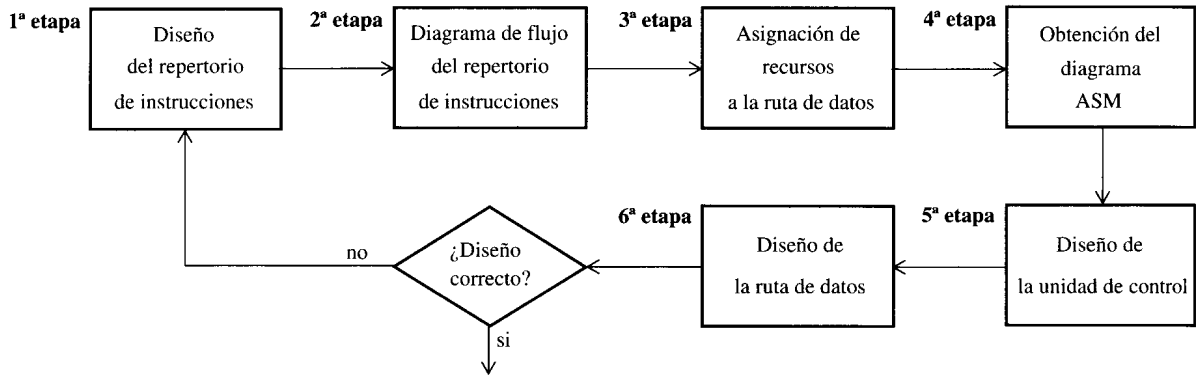


Figura 6.39: Etapas en el diseño del procesador

1) La primera etapa en el diseño del procesador debe ser necesariamente definir su *repertorio de instrucciones* especificando las operaciones que se realizan en cada una de ellas.
2) La segunda etapa consiste en describir, de forma no ambigua, las operaciones que se realizan en cada instrucción mediante un *diagrama de flujo del repertorio de instrucciones.*
3) La tercera etapa *asigna los recursos que se necesitan en la ruta de datos* para que pueda ser compatible con el diagrama de flujo deducido en la etapa anterior. Este proceso puede requerir varias iteraciones ya que el repertorio de instrucciones y la ruta de datos son interdependientes.
4) En la cuarta etapa, una vez definidos los recursos de la ruta de datos del procesador, se pasa a la *determinación del diagrama ASM del procesador* que divide cada instrucción del repertorio en una serie de ciclos de reloj y especifica las transferencias que se producen en cada uno de los ciclos.
5) Las dos últimas etapas son el *diseño de la unidad de control* y las *conexiones de la ruta de datos.*
6) Finalmente se analiza si el diseño resultante se puede considerar satisfactorio o no. En caso negativo habrá que iterar el diseño o algunas de sus etapas hasta lograr los objetivos perseguidos.

En la siguiente sección se presenta el diseño de un procesador elemental siguiendo la metodología que se acaba de describir.

Sección 6-5

Diseño de un procesador elemental

6.5.1 Especificación del procesador SIMPLE1

SIMPLE1 es un procesador elemental y académico que permite mostrar de forma sencilla los conceptos que se han introducido en este tema. En este apartado se exponen las características estructurales de SIMPLE1, un procesador ficticio extraordinariamente simplificado cuya única finalidad es mostrar paso a paso su diseño con lógica cableada. En una visión macroscópica un procesador se describe especificando los registros que son visibles a un programador y su repertorio de instrucciones.

Las instrucciones de SIMPLE1 ocupan una sola palabra de doce bits dividida en dos campos: el código de operación (*código_op*) y la dirección del operando. El formato de instrucciones es el representado en la Figura 6.40. En la parte superior se han numerado los bits de la palabra: la dirección del operando ocupa los bits 0 a 8 y el *código_op* está en los bits 9, 10 y 11.

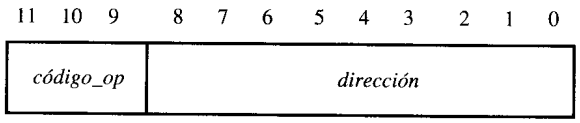


Figura 6.40: Formato de instrucción de SIMPLE1

El flujo de información dentro de SIMPLE1 se realiza por medio de un único bus de 12 bits. Este bus transmite tanto datos (12 bits) como direcciones (9 bits). Cuando transmite una dirección, los tres bits más significativos los rellena con ceros. Una parte importante de cualquier procesador son sus registros de trabajo. SIMPLE1 tiene los siguientes registros:

- *Registro contador de programa* (*PC*). El contenido de *PC* almacena la dirección de la siguiente instrucción que se va a ejecutar. Este registro tiene una longitud de 9 bits.
- *Registro de instrucción* (*IR*). El contenido de *IR* almacena la instrucción que se está ejecutando. Este registro tiene una longitud de 12 bits.
- *Registro de dirección de memoria* (*MAR*). El contenido de *MAR* especifica la próxima dirección de memoria de donde se va a leer o donde se va a escribir. Este registro tiene una longitud de 9 bits.
- *Registro de datos de memoria* (*MBR*). Este registro contiene el dato a escribir en la memoria o recibe el dato leído de la memoria. Este registro tiene una longitud de 12 bits.
- *Dos registros de trabajo A* y *B*. Estos dos registros tienen una longitud de 12 bits.

La unidad aritmético-lógica (ALU) de SIMPLE1 tiene dos entradas, que son operandos de doce bits, y una salida que es un resultado de doce bits. Las operaciones posibles son solamente dos:

- *Sumar* los dos operandos presentes en las entradas.
- *Restar* los dos operandos presentes en las entradas.

La memoria que se conecta a SIMPLE1 tiene una capacidad de 512 palabras de 12 bits cada una. La lectura de un dato se realiza colocando la dirección del dato a leer en el registro de direcciones de memoria (MAR) y activando la señal *R* de lectura; el dato leído se almacena en el registro de datos de memoria (MBR). Por otro lado, para la escritura de un dato en la memoria se necesita colocar el dato en el registro MBR, la dirección donde se va a almacenar en MAR y activar la señal *W* de escritura.

6.5.2 Repertorio de instrucciones

El repertorio de SIMPLE1 consta de siete instrucciones, identificadas por un código de operación de tres bits. En la Tabla 6.6 se muestran estas instrucciones, con su código en binario, su significado y un código simbólico ó nemotécnico para identificarlas de forma más sencilla. Cuatro de las instrucciones (LDA, STA, BR y BRN) hacen *referencia a memoria* (las dos primeras, a la dirección de un operando, las otras dos, a la dirección de la instrucción que la CPU debe ejecutar a continuación), y tres no (ADD, SUB y MAB).

SIMPLE1 sólo permite dos modos de direccionamiento: *direccionamiento implícito* (ADD, SUB y MAB) y *direccionamiento directo* (LDA y STA). Como se observa, lo que limita la capacidad máxima de memoria es la longitud de palabra y el formato de instrucciones elegido, en el que hay nueve bits para la dirección ($2^9 = 512$).

Las instrucciones de salto, BR y BRN, le indican a la CPU que la siguiente instrucción a ejecutar se encuentra en la dirección de la memoria principal indicada por el campo de dirección. BRN condiciona esta bifurcación a que el último resultado en la ALU haya sido negativo (el indicador de número negativo IN es el bit 11 del resultado). La razón de por qué el bit 11 del resultado refleja que el resultado es negativo es la siguiente. Las operaciones en la ALU son "sumar B a A" y "restar B de A" ("transferir A a B" no se puede considerar propiamente una operación de la ALU). Por tanto, en SIMPLE1 el resultado de la ALU siempre se introduce en el registro A, y la condición para el salto en BRN es equivalente a decir que el contenido del registro A es negativo (supuesto que las operaciones aritméticas de SIMPLE1 se efectúan en complemento a 2, el bit de signo vale uno si el numero es negativo).

Nemotécnico	Código binario	Instrucción	Acción
LDA x	LDA = 001	Carga directa	A ← M[x]
STA x	STA = 010	Almacenamiento directo	M[x] ← A
ADD	ADD = 011	Suma B a A	A ← A + B
SUB	SUB = 100	Resta B de A	A ← A - B
MAB	MAB = 101	Mueve A a B	B ← A
BR x	BR = 110	Salto incondicional a x	PC ← x
BRN x	BRN = 111	Salto a x si indicador negativo a 1	PC ← x si IN = 1

Tabla 6.6: Repertorio de instrucciones de SIMPLE1

6.5.3 Diagrama de flujo del repertorio de instrucciones

El repertorio de instrucciones que se acaba de definir especifica completamente el comportamiento de SIMPLE1 y por tanto es una descripción del procesador. Una de las formas utilizadas con frecuencia para representar el comportamiento es el *diagrama de flujo del repertorio de instrucciones*, que describe las fases de ejecución de todas las intrucciones. En la Figura 6.41 se muestra el diagrama de flujo para el repertorio de instrucciones de la Tabla 6.6.

Obsérvese que cada instrucción se ha especificado en dos partes. En la primera parte, que se aplica a todas las instrucciones, SIMPLE1 busca la instrucción en la memoria del computador, la introduce en el registro de instrucción IR e incrementa el contador de programa PC (fase de búsqueda).

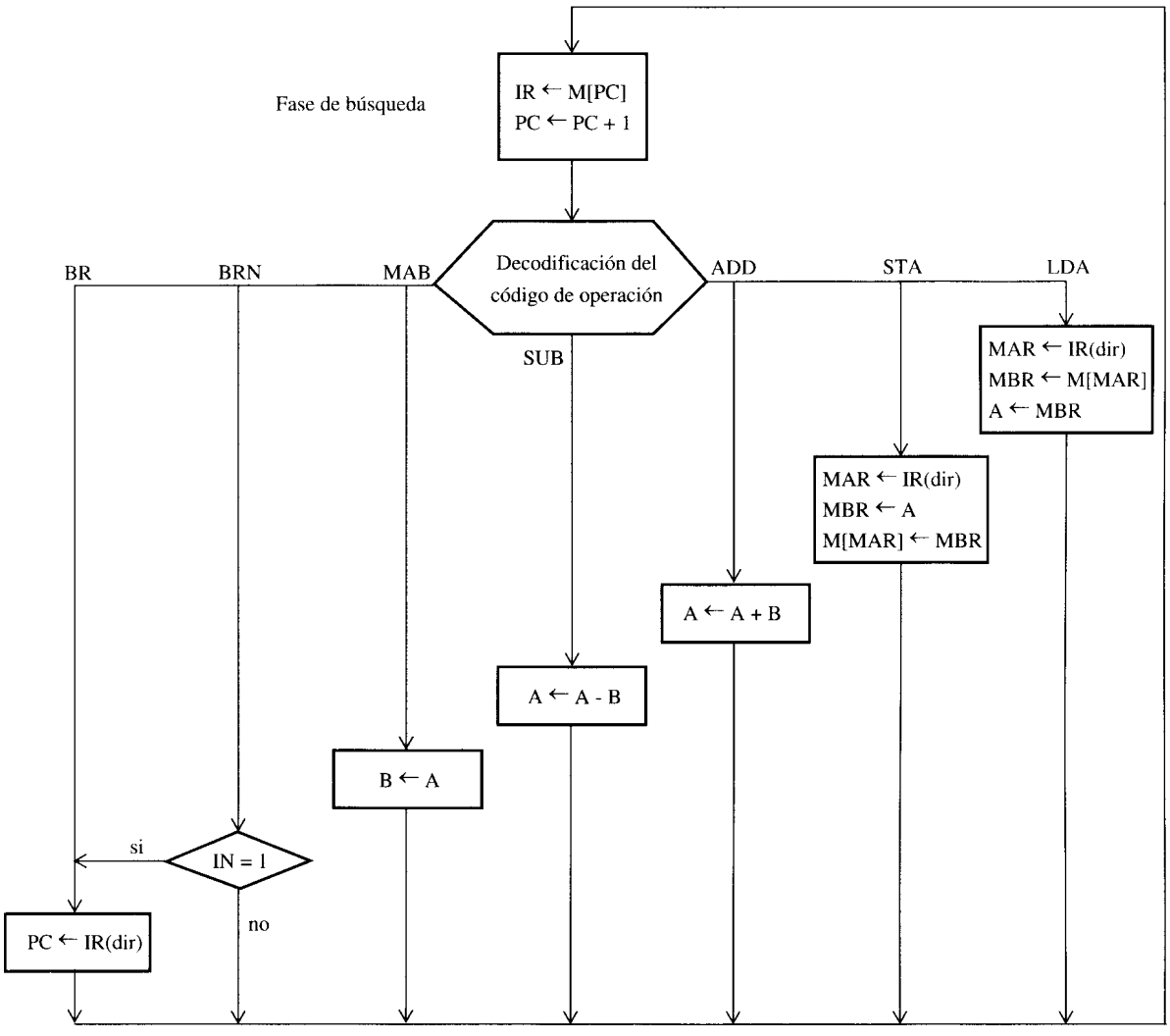


Figura 6.41: Diagrama de flujo del repertorio de instrucciones de SIMPLE1

En la segunda parte, se decodifica el código de operación de la instrucción, localiza el operando (en el caso de instrucciones de memoria o de bifurcación) y se ejecuta la operación especificada en su código de operación (fases de decodificación, búsqueda de los operandos y ejecución).

Este diagrama de flujo no incluye ningún detalle relativo a la arquitectura del computador salvo en lo que respecta a los elementos y registros que lo constituyen. Tampoco toma en consideración ninguna restricción temporal ni la duración de los ciclos de reloj. Su finalidad es simplemente proporcionar el orden en el que se han de ejecutar las microoperaciones especificadas para cada instrucción.

6.5.4 Asignación de recursos a la unidad de procesamiento o ruta de datos

Una vez que se ha completado el diagrama de flujo del repertorio de instrucciones, el siguiente paso en el diseño del procesador es asignarle sus recursos seleccionando su número, el tipo de las unidades que se van a incluir en su ruta de datos y las conexiones que hay entre los diferentes registros. En la Figura 6.42 se muestra el conjunto de componentes de que dispone SIMPLE1 y las transferencias de datos que están permitidas.

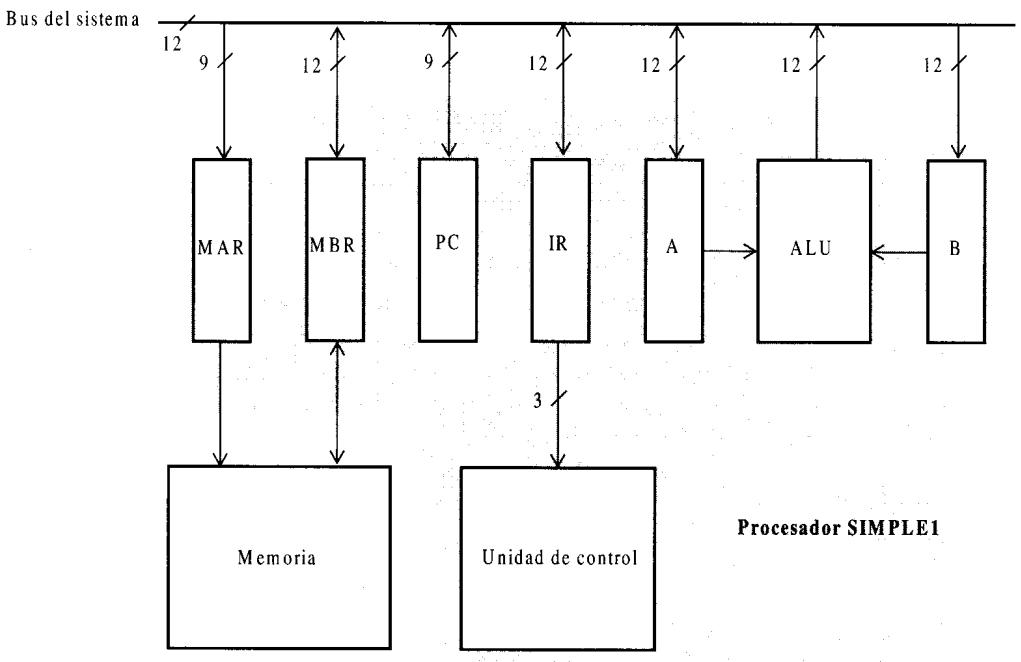


Figura 6.42: Asignación de recursos en el procesador SIMPLE1

En la Figura 6.42 no se han incluido aún ni las señales de control ni de condición. Se observa que todo el intercambio de información del procesador con la memoria se efectúa a través de los registros MAR y MBR. Esta restricción, como ya se comentó al explicar la fase de búsqueda de una instrucción, impone que para acceder a una instrucción en memoria habrá que transferir en primer lugar el contenido de PC a MAR.

6.5.5 Obtención del diagrama ASM del procesador

El siguiente paso en el diseño de SIMPLE1, es la obtención del diagrama ASM a partir de la información presente en el diagrama de flujo de la Figura 6.41 y la asignación de recursos realizada en la Figura 6.42. En la Figura 6.43 se muestra el diagrama ASM de SIMPLE1.

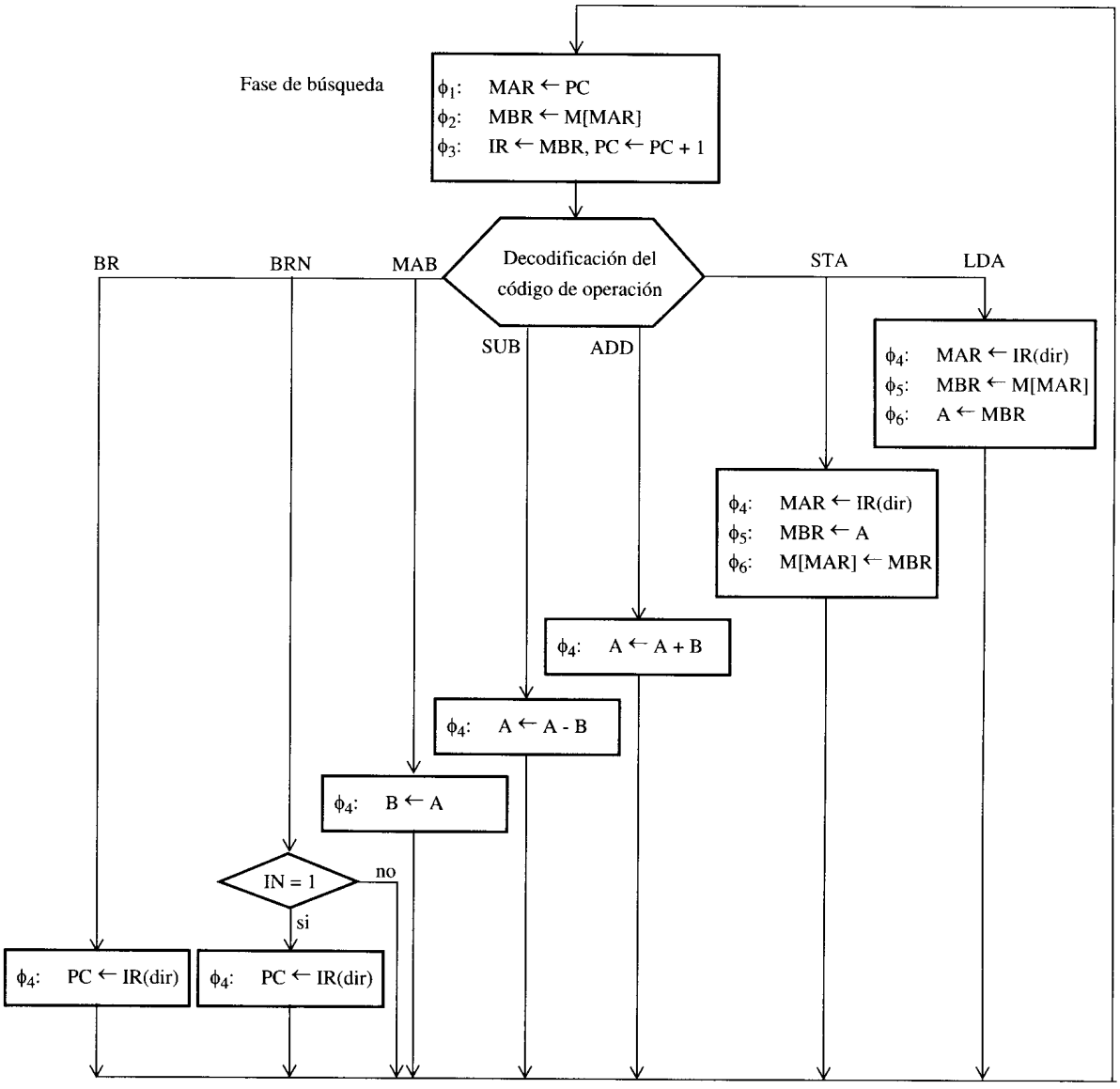


Figura 6.43: Diagrama ASM del procesador SIMPLE1

Para este caso tan sencillo se puede hacer de forma manual. Es muy importante resolver cualquier conflicto potencial que se pueda producir entre los datos y los recursos de que se dispone, asignando las operaciones o las transferencias de datos conflictivas a estados diferentes o a ciclos de reloj distintos. Así por ejemplo, es posible leer un valor almacenado en un registro y escribir un nuevo valor en ese registro durante el mismo ciclo de reloj puesto que las lecturas de registros se pueden realizar en cualquier instante de dicho ciclo que sea posterior al flanco de subida de la señal de reloj, mientras que las escrituras sólo se producen al final del ciclo de reloj en el siguiente flanco de subida de la señal de reloj. Sin embargo, no es posible escribir un valor en un registro y leer el nuevo valor en el mismo ciclo de reloj ya que la escritura, como se acaba de decir, sólo ocurre al final de dicho ciclo. Igualmente, dentro de un mismo ciclo, se puede leer o escribir en una posición de memoria, pero no se pueden hacer ambas cosas puesto que la memoria sólo tiene un decodificador de direcciones. Por idéntico motivo, el bus del sistema sólo se puede emplear una vez en un ciclo de reloj. Si se respetan estas restricciones, el proceso de transformación del diagrama de flujo del repertorio de instrucciones en un diagrama ASM es relativamente directo ya que son muy similares.

Todo lo que se ha hecho en la Figura 6.43 con respecto a la Figura 6.41 ha sido dividir el ciclo de ejecución de cada instrucción en varios subciclos para eliminar las dependencias entre recursos. Como se muestra en la Figura 6.43 la fase de búsqueda se completa en tres subciclos (ϕ_1, ϕ_2 y ϕ_3). Las microoperaciones asociadas a la fase de búsqueda son idénticas a las mostradas en la Tabla 6.3 y no precisan de ninguna aclaración adicional.

Las instrucciones LDA y STA precisan tres subciclos más (ϕ_4, ϕ_5 y ϕ_6). Considérese, por ejemplo, la instrucción LDA *x*. En el subciclo ϕ_4 el procesador transfiere el contenido del campo de dirección del registro IR al registro MAR ($MAR \leftarrow IR(dir)$). En el subciclo ϕ_5 el procesador lee el contenido de la posición de memoria referenciada por MAR y lo almacena en MBR ($MBR \leftarrow M[MAR]$). Finalmente en el subciclo ϕ_6 el procesador transfiere el contenido del registro MBR al registro *A* ($A \leftarrow MBR$). El resto de las instrucciones del repertorio sólo necesitan un subciclo. Para cada una de las microoperaciones que se detallan en el diagrama ASM se especifica el subciclo ϕ_i en que tiene lugar.

6.5.6 Diseño de la unidad de control

De forma genérica en la Figura 6.44 se muestra la unidad de control tal como se ha considerado hasta ahora. Las entradas de la unidad de control son el registro de instrucción IR, el reloj y las señales de condición. En las señales de condición cada bit individual normalmente tiene un significado concreto (por ejemplo indicación de rebose en la ALU, resultado negativo, etc). Las otras dos entradas, sin embargo, no son directamente utilizables por la unidad de control y necesitan un tratamiento previo.

En el caso del registro de instrucción, como el procesador interpreta un repertorio de instrucciones será necesario un circuito asociado a la unidad de control que identifique unívocamente a cada una de ellas, de tal forma que si la CPU está ejecutando, por ejemplo, la instrucción I_1, la unidad de control genera exclusivamente el conjunto de microórdenes que se necesitan para esa instrucción. El elemento responsable de esta función es el denominado *decodificador de instrucciones*, que es un circuito combinacional que tiene como entrada el código de instrucción almacenado en el registro de instrucción.

El generador de subciclos de la unidad de control produce una secuencia de pulsos repetitivos que se utilizan para activar las microórdenes en los instantes de tiempo que se necesitan en cada una de las fases que constituyen el ciclo de ejecución de una instrucción. El período de estos pulsos de reloj debe ser suficiente para permitir la propagación de las señales a través de los distintos caminos de datos que existan en la CPU.

Una forma de realizar esto, como ya se ha dicho, es dividir el ciclo de instrucción en un conjunto de subciclos ϕ_i, $i = 1, 2, ... n$ suficientes para que se pueda ejecutar todo el repertorio de instrucciones del procesador. Esta secuencia se repite periódicamente con la ejecución de cada instrucción. La forma más natural de implementar el generador de subciclos es con un contador módulo n (capaz de pasar al estado inicial desde cualquiera de los otros estados según sea la instrucción que se está ejecutando), donde n representa el máximo número de estados en cualquiera de los caminos que se puedan seguir en el diagrama ASM.

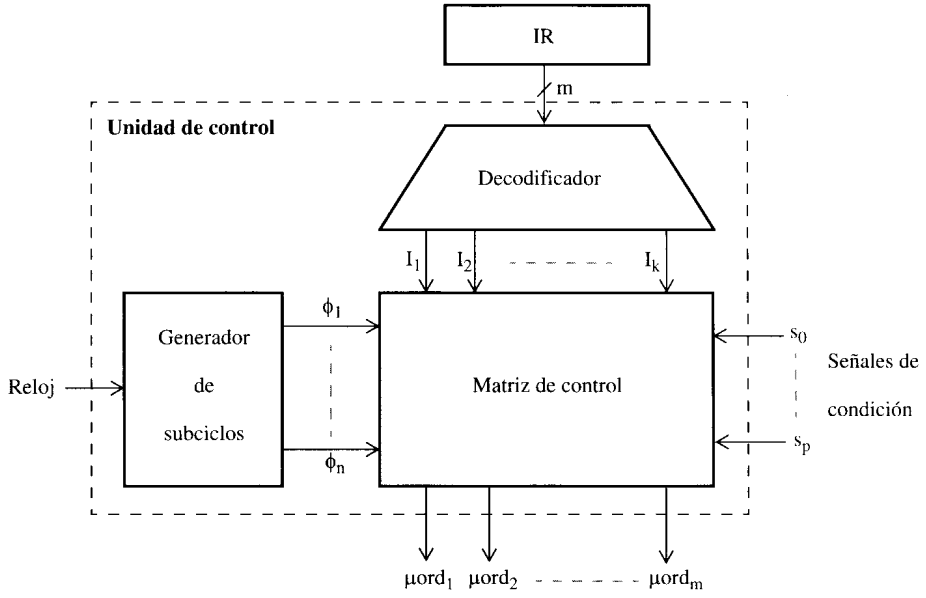


Figura 6.44: Unidad de control con decodificación de sus entradas

La *matriz de control* en la Figura 6.44 no es nada más que un circuito combinacional que genera las señales de control correspondientes a cada una de las microórdenes μord_i, $i = 1, 2, ... m$. De forma general, la ecuación lógica de una μord_i puede especificarse mediante una expresión booleana como suma de términos productos de la forma siguiente:

$$\mu ord_i = \sum_j (\phi_j \sum_q I_q[s_q])$$

donde ϕ_j son los subciclos en los que se necesitan activar la microorden, I_q representa la salida q-ésima del decodificador de instrucciones que identifica a una instrucción en particular y s_q las señales de condición que pueden ser necesarias para su ejecución (por ejemplo s_1 igual a 1 si el contenido del acumulador es cero). Los corchetes que rodean a s_q expresan que estas señales son opcionales y dependen por lo tanto de cada microorden específica.

Unidad de control de SIMPLE1

Estas ideas se pueden aplicar al caso concreto del procesador SIMPLE1 que se está diseñando en esta sección. En la Tabla 6.7 se muestra el conjunto de las dieciseis señales de control que tiene el procesador

SIMPLE1. Se han deducido mediante un análisis sistemático de todas las microoperaciones que se tienen en el diagrama ASM de la Figura 6.43 y teniendo en cuenta la arquitectura del procesador que se ha propuesto en la Figura 6.42. La mayor parte de los registros de SIMPLE1 tienen dos señales de control, una de carga (que se nombra anteponiendo la letra *C* al nombre del registro) y otra de habilitación (que se nombra anteponiendo la letra *H* al nombre del registro). Así la activación de la señal *CMBR* provoca la carga en el registro MBR del contenido del bus y la activación de la señal *HMBR* hace que el contenido del registro MBR pase al bus. Por ejemplo la microoperación ϕ_4: *MAR* ← *IR*, de la instrucción LOAD, requiere que se activen de forma simultánea las señales de control *HIR* (*Bus* ← *IR*) y *CMAR* (*MAR* ← *Bus*). Obsérvese que el registro *B* sólo dispone de una señal de carga *CB* y no tiene ninguna señal de habilitación. Esto se debe a la asignación de recursos que se ha realizado para el procesador SIMPLE1 (ver Figura 6.42) que no permite que el contenido de B pase al bus.

Señal de control	Microorden controlada
R	Leer de la memoria (MBR ← M[MAR])
W	Escribir en la memoria (M[MAR] ← MBR)
CMAR	Cargar el contenido del bus en MAR (MAR ← Bus)
HMBR	Habilitar el registro MBR (Bus ← MBR)
CMBR	Cargar el contenido del bus en MBR (MBR ← Bus)
HPC	Habilitar el registro PC (Bus ← PC)
CPC	Cargar el contenido del bus en PC (PC ← Bus)
IPC	Incrementar el contenido de PC (PC ← PC + 1)
HIR	Habilitar el registro IR (Bus ← IR)
CIR	Cargar el contenido del bus en IR (IR ← Bus)
HA	Habilitar el registro A (Bus ← A)
CA	Cargar el contenido del bus en A (A ← Bus)
CB	Cargar el contenido del bus en B (B ← Bus)
HALU	Habilitar la unidad aritmético-lógica
SUMA	Seleccionar la función de suma en la unidad aritmético-lógica
RESTA	Seleccionar la función de resta en la unidad aritmético-lógica

Tabla 6.7: Señales de control del procesador SIMPLE1

En la Figura 6.45 se muestra la unidad de control del procesador SIMPLE1. Puede observarse que sus entradas son el código de operación de la instrucción (los tres bits más significativos del registro IR), la señal de reloj y la señal de condición IN de resultado negativo. Las salidas son las dieciseis señales de control.

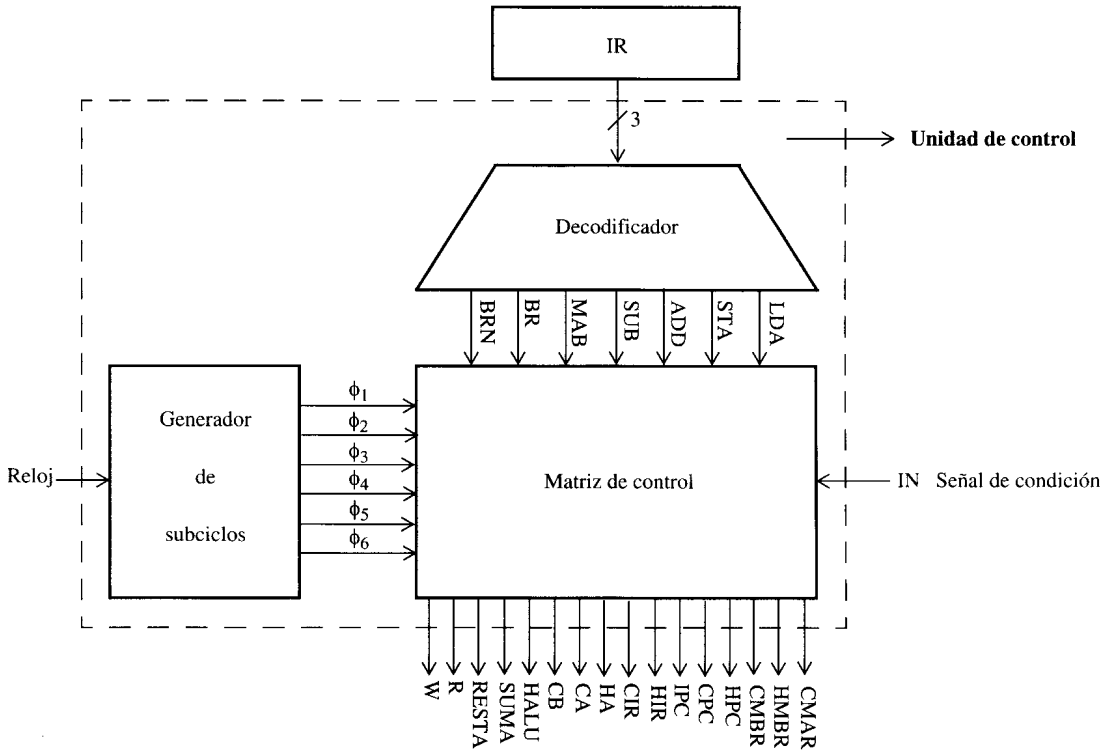


Figura 6.45: Unidad de control del procesador SIMPLE1 con decodificación de sus entradas

En la Figura 6.46 se muestra cómo se decodifican las instrucciones. El proceso es análogo a la operación del decodificador descrito en el apéndice A (ver sección A-3). Para cada uno de los siete códigos de operación, una y sólo una de las salidas está a 1. Por ejemplo, si durante la fase de búsqueda se carga en el registro IR el *código_op* de la instrucción STA (010), sólo estará a 1 la salida de la puerta AND correspondiente a este código de operación (salida 2), las restantes salidas del decodificador estarán a 0.

En la Figura 6.47 se muestra la división del ciclo de instrucción del procesador SIMPLE1 en los seis subciclos que se necesitan para su ejecución. Una forma sencilla de implementar el generador de subciclos es mediante un contador en anillo módulo-6 que se construye con seis elementos de memoria. Sincronizado por el reloj del sistema, primero activa su salida ϕ_1, luego ϕ_2 y así sucesivamente, de modo que tras activar ϕ_6 se vuelva a ϕ_1.

La elección de un contador módulo-6 es fácil de comprender, como ya se ha dicho, examinando el diagrama ASM de la Figura 6.43. Se observa que la fase de búsqueda de la instrucción en memoria requiere tres utilizaciones del bus y la ejecución de las instrucciones implica en el caso más complejo otras tres utilizaciones más del bus. En total se necesitan realizar seis accesos al bus, por lo que la interpretación y ejecución de una instrucción como máximo obliga a realizar seis acciones secuenciales. Cada acción se realiza en un estado sucesivo del contador.

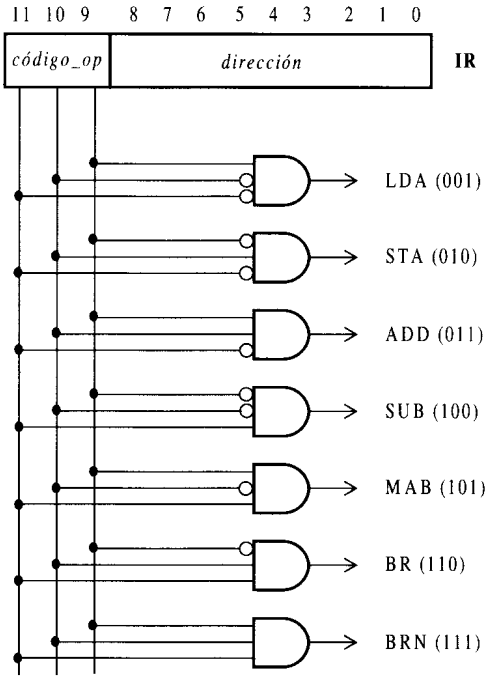


Figura 6.46: Decodificador de instrucciones del procesador SIMPLE1

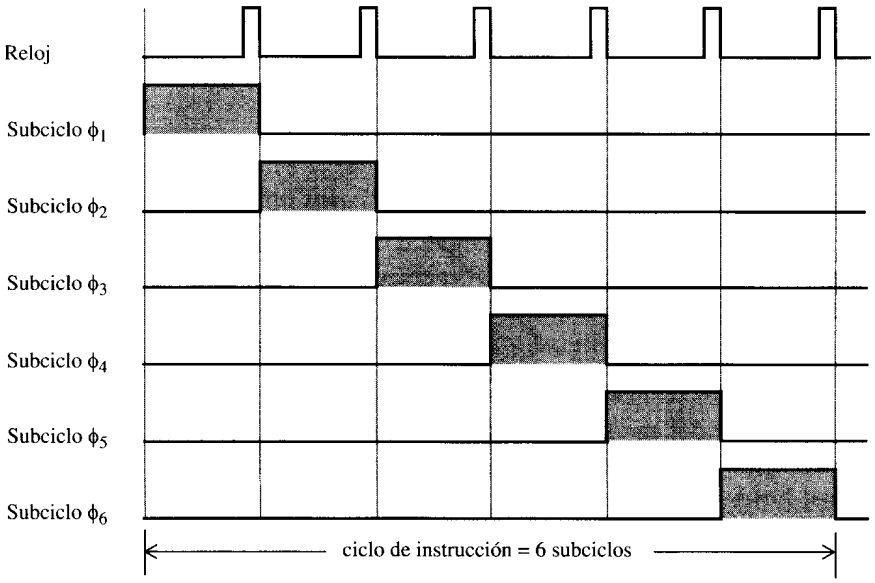


Figura 6.47: División del ciclo de instrucción del procesador SIMPLE1 en 6 subciclos

Para pequeños valores de m un contador en anillo módulo-m no requiere ninguna decodificación para la generación de los subciclos ya que son simplemente las salidas de los m elementos de memoria que contiene el contador. Básicamente es un registro de desplazamiento de m bits donde el estado i del contador es de la forma 00...010...0 con un 1 en el elemento de memoria i-ésimo y con 0's en los restantes m - 1 elementos de memoria. La cuenta i se incrementa a $i + 1$ mediante un desplazamiento de 1 bit del contenido del contador. Por ejemplo para el contador módulo-6 (ver Figura 6.48) de SIMPLE1 se requiere que pase a través de la siguiente secuencia de estados de forma cíclica:

100000 → 010000 → 001000 → 000100 → 000010 → 000001

Las salidas del elemento de memoria que se encuentra más a la derecha se conecta a las entradas del elemento de memoria más a la izquierda para formar así un camino cerrado o anillo alrededor del cual el 1 almacenado puede circular; de este hecho deriva el nombre de contador en anillo.

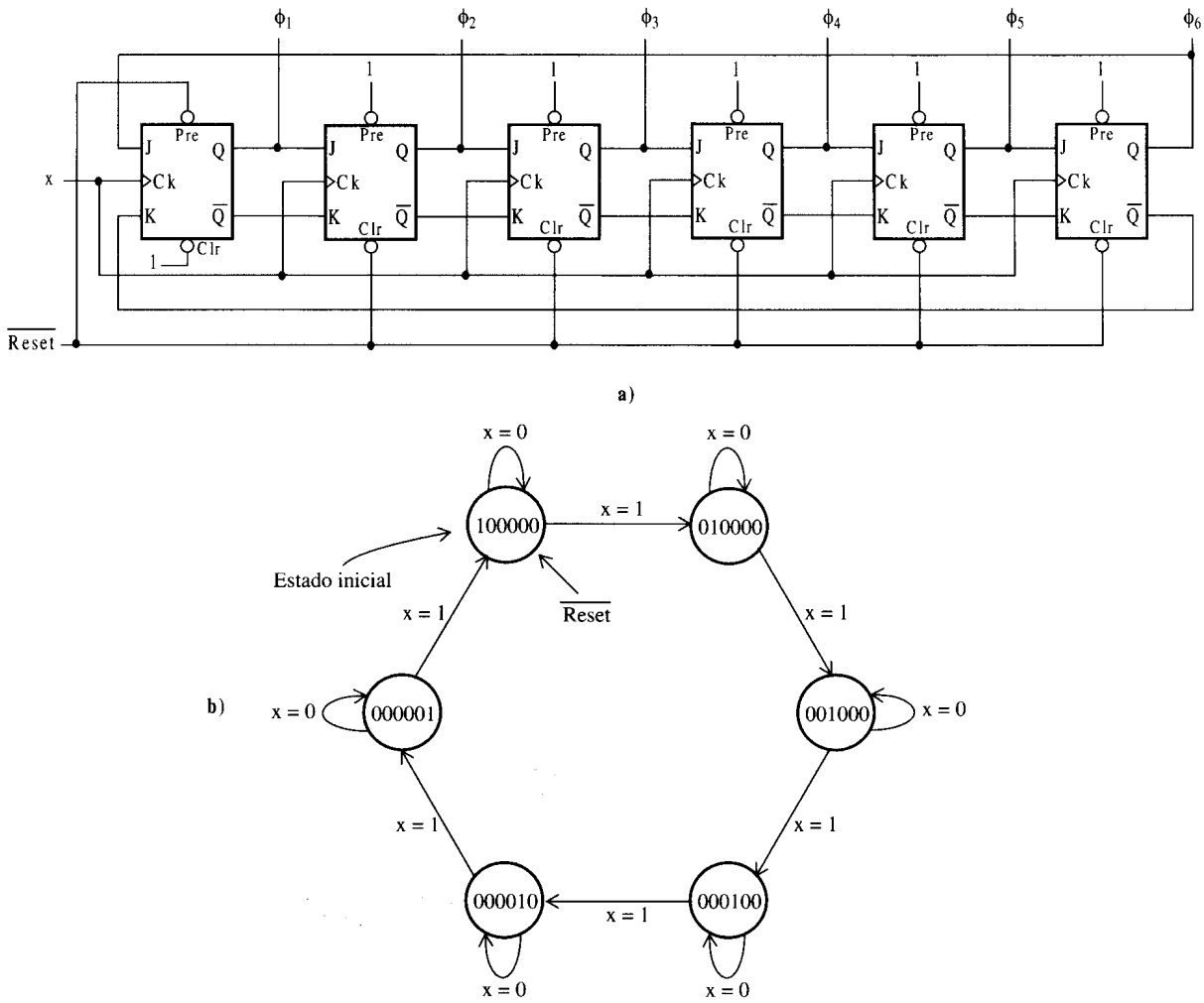


Figura 6.48: Contador en anillo módulo-6: a) Circuito lógico; b) Diagrama de estado

La Figura 6.48 muestra un contador en anillo módulo-6 que está compuesto de seis elementos de memoria *JK*. El contador se pone en su estado inicial 100000 activando la línea de entrada asíncrona, que pone a 1 el elemento de memoria más a la izquierda y los restantes a 0. Obsérvese que la entrada de reloj externa está conectada directamente a la entrada de reloj de cada elemento de memoria de forma que todos ellos cambian de estado simultáneamente.

Para completar el diseño de la unidad de control sólo queda por especificar el circuito combinacional de la *matriz de control*, para lo que se sigue un proceso sistemático a partir de la Tabla 6.8. Esta tabla se obtiene directamente del diagrama ASM de la Figura 6.43 al incluir las señales de control de la Tabla 6.7 que hay que activar para cada una de las microoperaciones.

	Acción	Microoperaciones	Señales de control
Búsqueda	Fase de búsqueda de la instrucción	ϕ_1: MAR ← PC; ϕ_2: MBR ← M[MAR]; ϕ_3: IR ← MBR, PC ← PC + 1;	ϕ_1: HPC, CMAR ϕ_2: R ϕ_3: HMBR, CIR, IPC
LDA x	Carga directa	ϕ_4: MAR ← IR(dir); ϕ_5: MBR ← M[MAR]; ϕ_6: A ← MBR;	ϕ_4: HIR, CMAR ϕ_5: R ϕ_6: HMBR, CA
STA dir	Almacenamiento directo	ϕ_4: MAR ← IR(dir); ϕ_5: MBR ← A; ϕ_6: M[MAR] ← MBR;	ϕ_4: HIR, CMAR ϕ_5: HA, CMBR ϕ_6: W
ADD	Suma B a A	ϕ_4: A ← A + B;	ϕ_4: SUMA, HALU, CA
SUB	Resta B de A	ϕ_4: A ← A - B;	ϕ_4: RESTA, HALU, CA
MAB	Mueve A a B	ϕ_4: B ← A;	ϕ_4: HA, CB
BR x	Salto incondicional a x	ϕ_4: PC ← IR(dir);	ϕ_4: HIR, CPC
BRN x	Salto a x si indicador negativo a 1	ϕ_4: PC ← IR(dir) (si IN = 1);	ϕ_4: si IN = 1: HIR, CPC

Tabla 6.8: Señales de control que hay que activar en cada microoperación

Esta misma información se puede poner de forma mas sistemática. Para ello, se construye una nueva tabla (ver Tabla 6.9) que tiene por filas las instrucciones que se han de ejecutar más la fase de búsqueda que es común a todas ellas y por columnas las dieciseis señales de control.

En el caso de que una señal de control deba activarse durante una instrucción, la celda correspondiente se rellena con el subciclo ϕ_i del contador en el que deba estar activa. Así, por ejemplo, la ejecución de la instrucción LDA requiere la activación de las señales de control *CMAR* y *HIR* en el subciclo ϕ_4, de *R* en el subciclo ϕ_5, y de *HMBR* y *CA* en el subciclo ϕ_6.

Una vez obtenida dicha tabla se puede obtener la función lógica de cada señal de control. Esta se expresa como la función OR de todos los instantes en que debe activarse, que a su vez son la función AND de la instrucción (salida del decodificador) con el subciclo en que debe estar activo (salida del generador de subciclos). En el caso del procesador SIMPLE1 se obtienen las siguientes ecuaciones lógicas:

	IPC	CPC	HPC	CMAR	R	W	CMBR	HMBR	CIR	HIR	CA	HA	SUMA	RESTA	HALU	CB
Búsqueda	ϕ_3		ϕ_1	ϕ_1	ϕ_2			ϕ_3	ϕ_3							
LDA				ϕ_4	ϕ_5			ϕ_6		ϕ_4	ϕ_6					
STA				ϕ_4		ϕ_6	ϕ_5			ϕ_4		ϕ_5				
ADD											ϕ_4		ϕ_4		ϕ_4	
SUB											ϕ_4			ϕ_4	ϕ_4	
MAB												ϕ_4				ϕ_4
BR		ϕ_4								ϕ_4						
BRN		ϕ_4 IN								ϕ_4 IN						

Tabla 6.9: Matriz de instantes de activación de las señales de control para cada instrucción de SIMPLE1

$$IPC = \phi_3$$

$$CPC = \phi_4 BR + \phi_4 IN\ BRN$$

$$HPC = \phi_1$$

$$CMAR = \phi_1 + \phi_4 LDA + \phi_4 STA$$

$$R = \phi_2 + \phi_5 LDA$$

$$W = \phi_6 STA$$

$$CMBR = \phi_5 STA$$

$$HMBR = \phi_3 + \phi_6 LDA$$

$$CIR = \phi_3$$

$$HIR = \phi_4 LDA + \phi_4 STA + \phi_4 BR + \phi_4 IN\ BRN$$

$$CA = \phi_6 LDA + \phi_4 ADD + \phi_4 SUB$$

$$HA = \phi_5 STA + \phi_4 MAB$$

$$SUMA = \phi_4 ADD$$

$$RESTA = \phi_4 SUB$$

$$HALU = \phi_4 ADD + \phi_4 SUB$$

$$CB = \phi_4 MAB$$

En la Figura 6.49 se muestra cómo a partir de estas ecuaciones lógicas, se obtiene de forma directa la matriz de control. Todo lo que hay que hacer es traducir las funciones correspondientes en un circuito combinacional con puertas AND y OR.

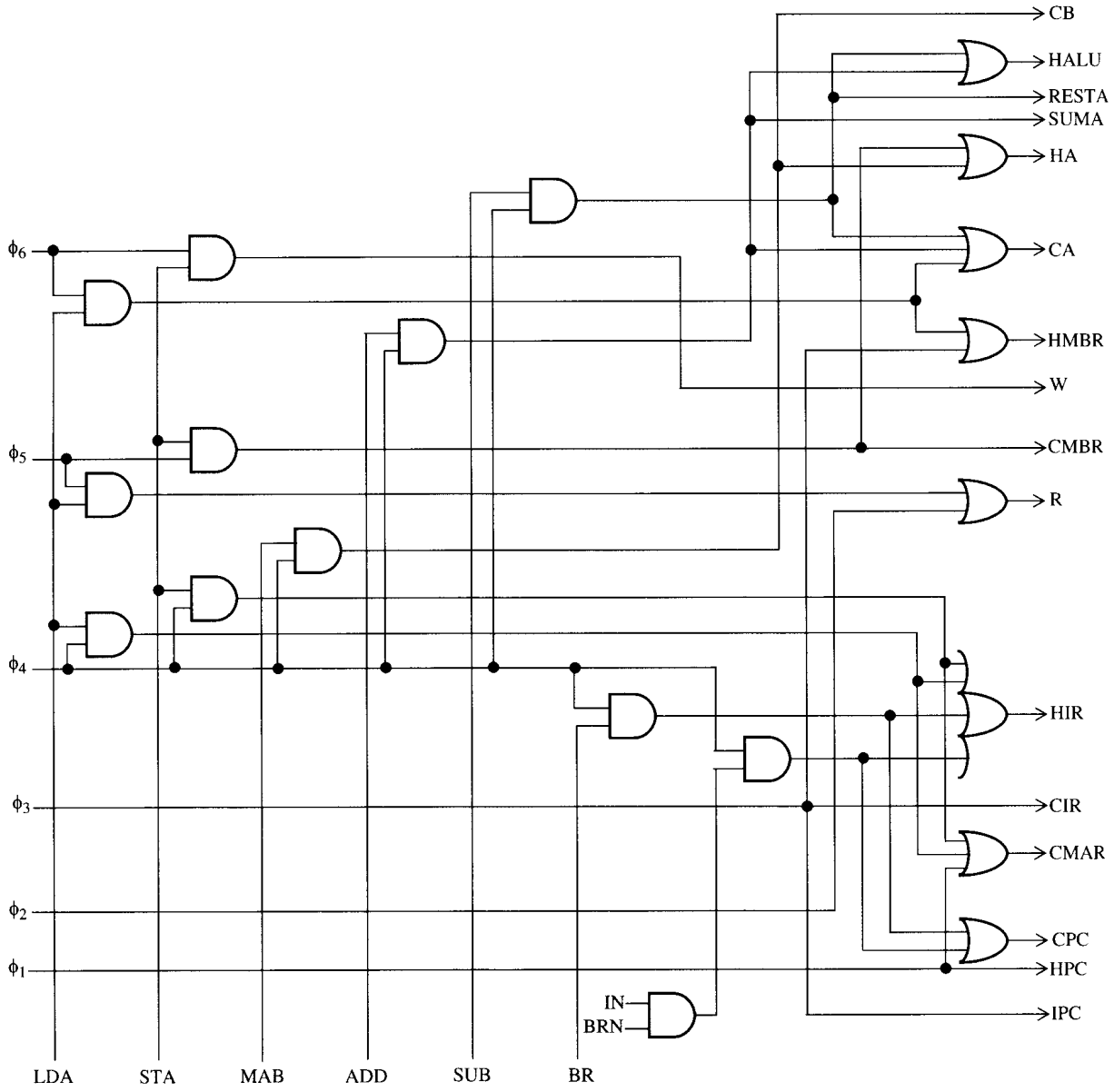


Figura 6.49: Matriz de control del procesador SIMPLE1

6.5.7 Diseño de la unidad de procesamiento o ruta de datos

El paso final en el diseño consiste en interconectar todos los elementos insertando las señales de control que genera la unidad de control en los recursos de cálculo de la ruta de datos. La Figura 6.50 muestra el esquema final del procesador SIMPLE1 con lo que queda completado su diseño.

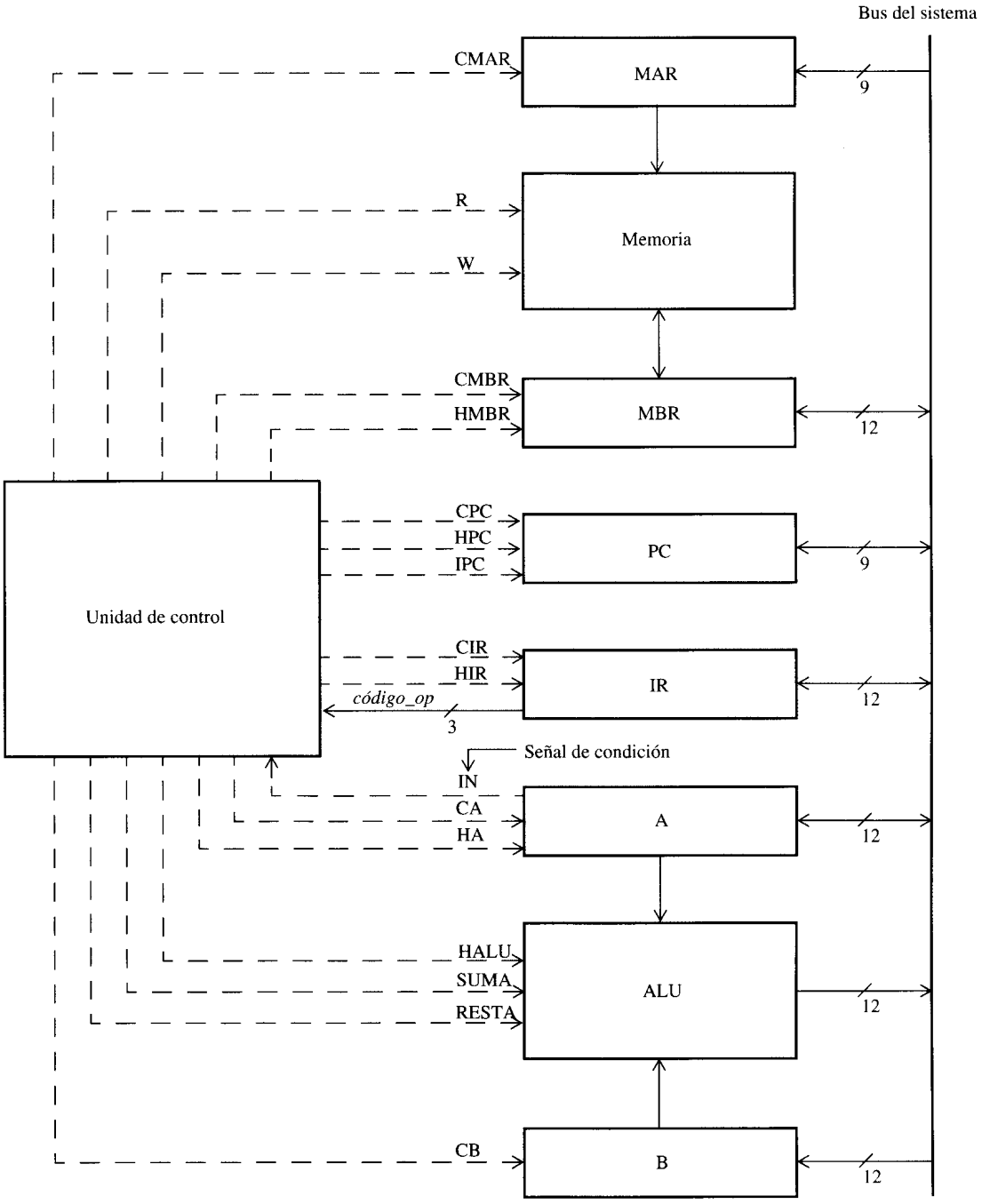


Figura 6.50: Procesador SIMPLE1 y su conexión con la memoria

Sección 6-6

Conclusiones

En este tema se han aplicado las técnicas para la síntesis de sistemas digitales a nivel de transferencias entre registros al diseño de procesadores de propósito general que quedan definidos por el repertorio de instrucciones que son capaces de ejecutar. Se ha efectuado un análisis comparativo en el que se han considerado distintos tipos de instrucciones, con sus campos y modos de direccionamiento y se ha puesto de manifiesto la íntima relación que hay entre el diseño del repertorio de instrucciones y la arquitectura seleccionada para el procesador.

Generalmente, las CPU´s incluyen varios centenares de instrucciones en sus repertorios, con formatos y modos de direccionamiento diversos y complicados. Una tendencia alternativa es la de los procesadores RISC, en los que el número de instrucciones y modos es reducido, el formato de instrucciones es regular y sólo se accede a la memoria principal para leer o escribir. Las instrucciones de procesamiento tienen los operandos y el resultado en registros de la CPU.

Una instrucción del lenguaje máquina de un computador se ejecuta mediante una secuencia de microoperaciones en su CPU. La unidad de control de la CPU es un sistema secuencial cuya misión es determinar el orden en el que los distintos componentes de la unidad de procesamiento o ruta de datos deben operar para que se efectúe la ejecución de una instrucción que se ha dividido en cuatro fases:

- Búsqueda de la instrucción
- Decodificación de la instrucción
- Búsqueda de los operandos
- Ejecución de la instrucción

La unidad de control recibe información sobre la instrucción que debe ser ejecutada a través de su código de operación y de las señales de condición generadas en la ruta de datos en la ejecución de la instrucción anterior. En función de esta información, la unidad de control actúa sobre los elementos de la ruta de datos de distinta forma, permitiendo la carga de sus registros, controlando los caminos que siguen los operandos, indicando a la ALU si debe o no operar y a la memoria si se debe realizar una operación de lectura o de escritura.

En este tema se han expuesto de forma sistemática las etapas que constituyen el ciclo de diseño de un procesador (unidad de control + ruta de datos). El punto de partida son las especificaciones que debe satisfacer. Se define el diagrama de flujo del repertorio de instrucciones y a partir del mismo se deduce el diagrama ASM que permite obtener la arquitectura del procesador.

Finalmente se ilustran estas ideas realizando paso a paso el diseño completo de SIMPLE1, un procesador sencillo introducido con fines exclusivamente académicos.

Sección 6-7 Problemas

1) Una CPU dispone de una ALU que puede realizar la siguiente función aritmética:

$$z = x + y \qquad z = \overline{x} \qquad z = \overline{y}$$

donde x e y son las entradas de la ALU, z es la salida y $\overline{x}$ e $\overline{y}$ representan los complementos a 1 de x e y respectivamente. Se supone que x e y están representados en complemento a 2. Expresar la secuencia de microoperaciones que debe efectuar la unidad de control para realizar la operación de restar.

2) En la Figura 6.51 se muestra la estructura de una CPU. Los puntos de control se representan mediante un pequeño círculo con una etiqueta c_i que identifica a la señal de control correspondiente. En la Tabla 6.10 se da una relación de las microoperaciones que controlan cada señal c_i.

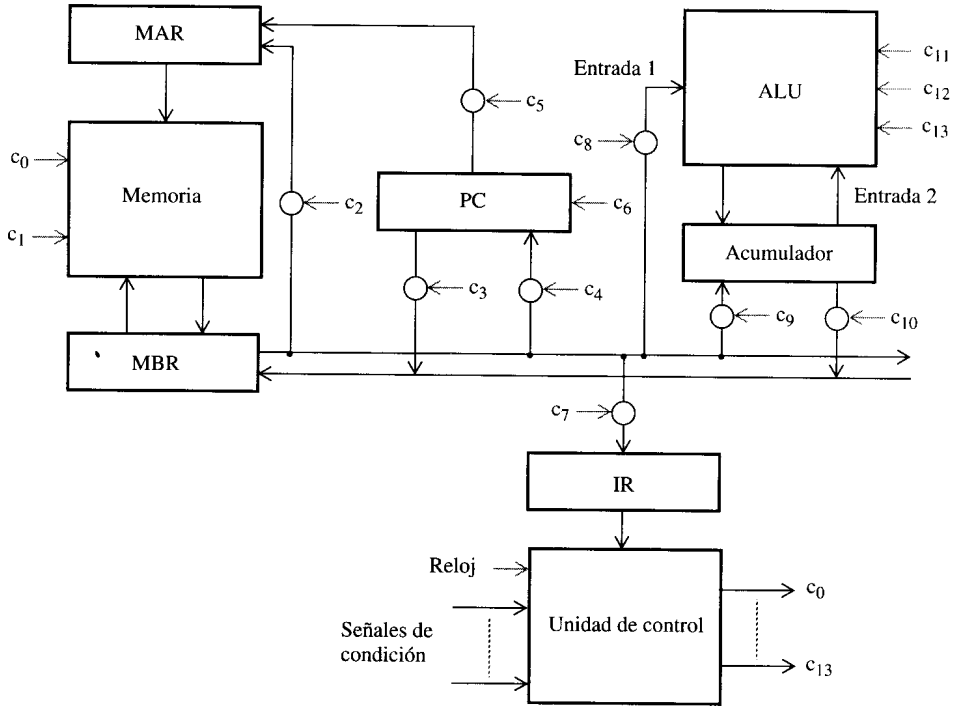


Figura 6.51: Caminos de datos y puntos de control de la CPU

Señal de control	Microoperación controlada
c_0	Leer de la memoria
c_1	Escribir en la memoria
c_2	Transferir el contenido de MBR a MAR (MAR ← MBR)
c_3	Transferir el contenido de PC a MBR (MBR ← PC)
c_4	Transferir el contenido de MBR a PC (PC ← MBR)
c_5	Transferir el contenido de PC a MAR (MAR ← PC)
c_6	Incrementar en 1 el contenido de PC (PC ← PC + 1)
c_7	Transferir el contenido de MBR a IR (IR ← MBR)
c_8	Transferir el contenido de MBR a la Entrada 1 de la ALU (Entrada 1← MBR)
c_9	Transferir el contenido de MBR al Acumulador (AC ←MBR)
c_{10}	Transferir el contenido del Acumulador a MBR (MBR ← AC)
c_{11}	Sumar el contenido del Acumulador a la Entrada 1 (AC ← AC + Entrada 1)
c_{12}	AND lógico del contenido del Acumulador con la Entrada 1 (AC ← AC AND Entrada 1
c_{13}	Complementar el contenido del Acumulador (AC ← $\overline{AC}$)

Tabla 6.10: Señales de control de la CPU de la Figura 6.51

Obtener la secuencia de microoperaciones y las señales de control que hay que activar para que la CPU de la Figura 6.51 ejecute el repertorio de instrucciones que se muestra en la Tabla 6.11.

Nemotécnico	Instrucción	Significado
LDA x	Cargar acumulador	AC ← M[x]
STA dir	Almacenar acumulador	M[x] ← AC
ADD x	Sumar al acumulador	AC ← AC + M[x]
AND x	AND con el acumulador	AC ← AC AND x
CAA	Complementar el acumulador	AC ← $\overline{AC}$
BR x	Salto incondicional a x	PC ← x
BRA x	Salto a x si acumulador = 0	PC ← x si AC = 0

Tabla 6.11: Repertorio de instrucciones de la CPU de la Figura 6.51

3) Para la CPU con bus interno de la Figura 6.52 escribir la secuencia de microoperaciones que se necesitan para sumar un número con el acumulador cuando el número es:

   a) Un operando inmediato

   b) Un operando con direccionamiento directo

   c) Un operando con direccionamiento indirecto

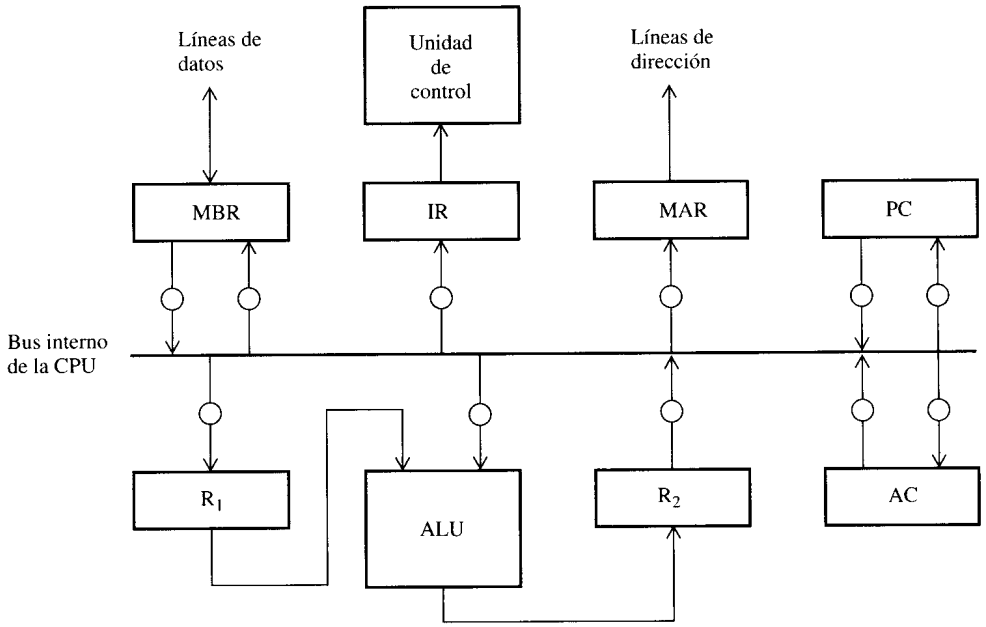


Figura 6.52: Estructura de la CPU con un bus interno

4) Escriba una secuencia de instrucciones que permita calcular el valor de la siguiente expresión:

$$a := [(a \times b + c) - d / f] / (a \times d) + b$$

utilizando:

   a) Instrucciones de tres direcciones

   b) Instrucciones de dos direcciones

   c) Instrucciones de una dirección (procesadores con acumulador)

   d) Instrucciones de cero direcciones (procesadores con pila)

Suponga que no hay banco de registros en el procesador. En el caso del apartado c) tan sólo se dispone de un acumulador. Indique en cada caso el número de instrucciones y de accesos a memoria.

5) Resuelva el problema anterior suponiendo que hay un banco de registros. Tanto para las direcciones del banco de registros como para las de memoria utilice:

   a) Instrucciones de una dirección

b) Instrucciones de dos direcciones

c) Instrucciones de tres direcciones

En cada caso minimice el número de direcciones de memoria

6) La mayoría de los procesadores disponen de una instrucción que hace pararse a la máquina, es decir, que no se vuelva a la fase de búsqueda de una instrucción en el ciclo que realiza de forma continuada la unidad de control. Amplíe el diseño del procesador SIMPLE1 introduciendo en su repertorio de instrucciones HALT, cuyo objetivo es detener la ejecución del programa que se esté realizando. Como el campo *código_op* de SIMPLE1 tiene tres bits y sólo se habían definido siete instrucciones es posible codificar la instrucción HALT (código de operación = 0_8) sin necesidad de modificar el resto.

7) Dada la arquitectura del procesador SIMPLE1 de la Figura 6.42 se pide:

a) Indicar las microoperaciones y las correspondientes señales de control que hacen falta para realizar B $\leftarrow$ M[x], es decir, el contenido de una dirección de memoria pasa al registro B.

b) ¿Cómo se puede efectuar la acción anterior haciendo uso del repertorio de instrucciones de la Tabla 6.6?

c) Determinar las modificaciones que son necesarias en la unidad de control, para ampliar el repertorio de instrucciones con la instrucción:

LDB x (B $\leftarrow$ M[x])

d) Si el dato que se quiere transferir al registro B está en la posición de memoria 12510, expresar en binario cómo estaría almacenada dicha instrucción máquina en memoria. Justificar cada bit o conjunto de bits en dicha palabra.

8) En la arquitectura del procesador SIMPLE1 de la Figura 6.42 se pretende prescindir del registro de datos de memoria (MBR), conectando la entrada/salida de datos de la memoria al bus del sistema.

a) ¿Qué otro elemento de la Figura 6.42 se vería afectado por dicha modificación?

b) ¿Qué cambios se introducen en el repertorio de instrucciones de la Tabla 6.6?

c) Rediseñar aquellos elementos de la unidad de control (ver Figura 6.45) que se ven afectados por la modificación propuesta.

9) Sin necesidad de modificar la arquitectura del procesador SIMPLE1 de la Figura 6.42 ¿qué cambios son necesarios realizar en la unidad de control para sustituir la instrucción BRN (bifurcar si negativo) por la instrucción BRNN (bifurcar si no negativo)?

10) Modificar el diseño del generador de subciclos del procesador SIMPLE1 (ver Figura 6.48) de forma que sea capaz de bifurcar al estado inicial desde cualquiera de los otros estados dependiendo de la instrucción que se está ejecutando (por ejemplo como la instrucción LDA x finaliza en el subciclo ϕ_4, en este caso el contador módulo-6 debería pasar al estado ϕ_1 sin necesidad de ir a los estados ϕ_5 y ϕ_6).

11) En la Tabla 6.12 se muestra el repertorio de instrucciones de un procesador de una dirección (procesador con acumulador). En la tabla se indica el nemotécnico de cada instrucción, su código binario y la acción

que realiza. Se supone que el formato de las instrucciones es idéntico al que se ha utilizado en el diseño del procesador SIMPLE1 (ver Figura 6.40).

Siguiendo la metodología explicada a lo largo del tema se pide diseñar este procesador analizando la alternativa que se proponga en relación con otras posibilidades que se pueden presentar.

Nemotécnico	Código binario	Instrucción	Significado
STA x	LDA = 000	Almacena el acumulador en la dirección x	M[x] ← AC
LDA x	LDA = 001	Carga en el acumulador el contenido de x	AC ← M[x]
ADD x	ADD = 010	Suma en el acumulador el contenido de x	AC ← AC + M[x]
BR x	BR = 011	Salto incondicional a x	PC ← x
BZ x	BZ = 100	Salto a x si acumulador es 0	PC ← x si AC = 0
CLR	CLR = 101	Pone a 0 el acumulador	AC ← 0
DEC	DEC = 110	Decrementa en 1 el acumulador	AC ← AC - 1
HALT	HALT = 111	Para la ejecución del programa	

Tabla 6.12: Repertorio de instrucciones del procesador del problema 11

Tema 7

Microprogramación

La unidad de control de un procesador controla la ejecución de las diferentes microoperaciones (μ_{op}) de su unidad de procesamiento o ruta de datos. Como regla general, la unidad de control envía un conjunto de microórdenes (μ_{ord}) por cada μ_{op}. Estas microórdenes son señales lógicas que permiten o impiden la realización de cada microoperación según su valor sea 1 ó 0. En cada instante habrá entonces un conjunto de μ_{ord}'s activadas (su valor es 1) y otro conjunto de μ_{ord}'s desactivadas (su valor es 0). En función de los resultados de las microoperaciones ejecutadas en el instante presente la unidad de control debe igualmente escoger las microórdenes que serán generadas en el instante siguiente.

Se dice que un procesador está microprogramado cuando las informaciones generadas por su unidad de control se almacenan en una memoria, la *memoria de control* (también denominada *memoria de microprograma* o *micromemoria*). Cada posición *i* de la memoria de control contendrá una microinstrucción $\mu I(i)$. Un *microprograma* μP no es nada más que una serie ordenada de *p* microinstrucciones:

$$\mu P = [\mu I(1), \mu I(2), \ldots, \mu I(p)] \tag{7.1}$$

El microprograma indica entonces las microinstrucciones que deben ser ejecutadas por la ruta de datos y su orden de ejecución (secuencia). Una unidad de control microprogramada debe cumplir las tres condiciones siguientes:

1) Ha de tener una memoria suficientemente grande para almacenar todos los microprogramas correspondientes a todas las instrucciones máquina de su repertorio.
2) Ha de asociar a cada instrucción la dirección de comienzo de su microprograma.
3) Ha de leer las sucesivas microinstrucciones del microprograma en curso y bifurcar a uno nuevo cuando termina el que se está ejecutando.

Los requisitos segundo y tercero se pueden resolver mediante los denominados *secuenciamientos explícito* e *implícito*. En el secuenciamiento explícito cada microinstrucción $\mu I(i)$ contiene la dirección de la microinstrucción siguiente además de las μ_{ord}'s correspondientes.

$$\mu I(i) = [d(\mu I(i)), \mu_{ord1}(i), \mu_{ord2}(i), \ldots, \mu_{ordm}(i)] \tag{7.2}$$

donde $d(\mu I(i))$ indica donde se encuentra la próxima microinstrucción que se tiene que ejecutar y μ_{ordj} es la microorden j de la unidad de control microprogramada. Las m μ_{ord}'s de $\mu I(i)$ deben ser independientes entre sí para poderse ejecutar simultáneamente durante el mismo período de reloj. Por este motivo hace falta una memoria de control de gran tamaño. En el secuenciamiento implícito, las microinstrucciones $\mu I(i)$ se disponen de forma consecutiva en la memoria, y se ejecutan en el orden en que están almacenadas, por lo que la memoria de control es de menor tamaño al no utilizarse el campo $d(\mu I(i))$ que apunta a la siguiente μI.

$$\mu I(i) = [\mu_{ord1}(i), \mu_{ord2}(i), ..., \mu_{ordm}(i)] \tag{7.3}$$

En este caso los requisitos segundo y tercero necesitan un mecanismo adicional para resolver el problema planteado. La solución es utilizar una microinstrucción especial que indica la bifurcación. Conviene advertir sin embargo que tanto en (7.2) como en (7.3) se podría haber definido una microinstrucción como un conjunto de microperaciones y no de microórdenes que se pueden ejecutar simultáneamente ya que como se ha indicado anteriormente una microoperación no es nada más que un conjunto de microórdenes (ver sección 5-3). Por este motivo a lo largo del texto se utilizarán ambas visiones de considerar una microintrucción como conjunto de microperaciones o de microórdenes, según convenga, ya que en último término son equivalentes. La ejecución de una microinstrucción requiere una secuencia de *búsqueda - decodificación - ejecución*. Esta secuencia, análoga pero más simple que la que se da en las instrucciones máquina, debe ser controlada por una nueva unidad de control. De una forma recursiva se podría ahora considerar que esta nueva y más sencilla unidad de control se realiza también con técnicas de microprogramación. La solución de este problema recursivo obliga a que la unidad de control final se realice mediante lógica de tipo cableado.

La microprogramación se puede considerar, según la idea original expuesta por el prof. M. V. Wilkes de la Universidad de Cambridge en 1950, *como un método sistemático para diseñar la unidad de control de cualquier sistema digital*. Sin embargo esta primera concepción de la microprogramación como un procedimiento diferente al tradicionalmente utilizado para el diseño de la unidad de control con lógica cableada, ha evolucionado llegando a convertirse en una alternativa no sólo de diseño sino de realización.

En la década de los 50 el concepto de la microprogramación sólo generó un relativo interés desde el punto de vista teórico. El principal motivo para este hecho estuvo en el alto coste que suponía su realización debido a la limitada tecnología de las memorias disponibles. En la mitad de la década de los 60 se produce un hecho fundamental para el desarrollo de la microprogramación, cuando el fabricante de computadores IBM la utiliza en la mayor parte de sus modelos de la *serie 360*, para el diseño de la unidad de control.

Hoy día la microprogramación suele aplicarse a los computadores de tamaño medio; en los pequeños no resulta rentable por su complejidad y en los grandes es demasiado lenta. Conceptualmente es muy simple, y permite hacer modificaciones y ampliaciones de manera sencilla. Las instrucciones complejas, de muchos periodos de duración, se realizan fácilmente, sin dificultades hardware. También es posible con la microprogramación que un computador sea capaz de ejecutar diferentes repertorios de instrucciones (sólo hay que modificar el contenido de la memoria de control). Esta característica permite *emular* en un único computador las prestaciones de distintos juegos de instrucciones.

En este tema se presentan los fundamentos de la microprogramación. En primer lugar se introduce el modelo original propuesto por M. Wilkes. A continuación se presentan las diferentes alternativas que se han propuesto para el diseño de una unidad de control microprogramada. El tema finaliza con dos ejemplos de diseño de unidad de control microprogramada: el multiplicador binario analizado en el tema 5 y SIMPLE2 que es un sencillo computador que dispone de una unidad de control microprogramable ($uc\mu P$).

Sección 7-1

Modelo original de Wilkes

La idea base de Wilkes fue considerar, como ya se ha visto anteriormente, que cada instrucción máquina del repertorio de un computador se puede ver como un conjunto de microórdenes que especifican las transferencias de información entre los diferentes componentes de su parte operacional. Activando los puntos de control necesarios en estos módulos funcionales se controla el flujo de datos entre ellos.

Una matriz de núcleos de ferritas con selección lineal (que era la tecnología de memoria que se disponía en esos momentos) fue el esquema utilizado en el primer modelo de Wilkes. Esta matriz de núcleos de ferritas estaba permanentemente cableada como un conjunto de puertas OR que pueden activar a un subconjunto de líneas de control que forman la salida de la matriz. En la Figura 7.1 se muestra el esquema original de Wilkes.

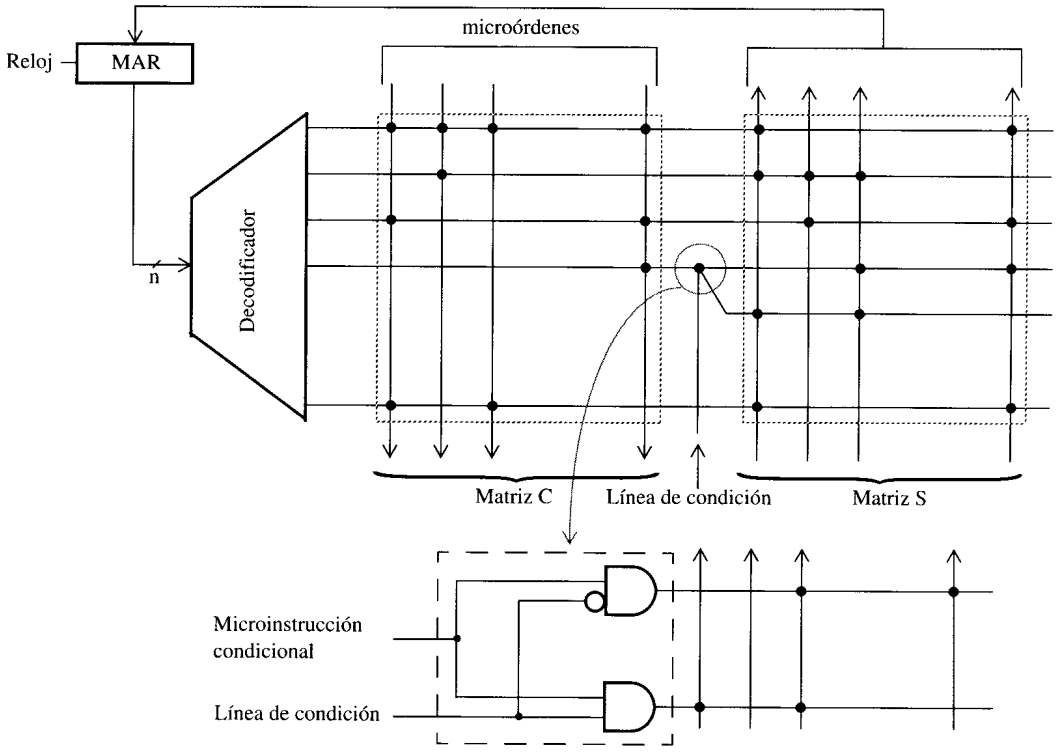


Figura 7.1: Modelo original propuesto por Wilkes

El sistema propuesto constaba de dos matrices, la *matriz de control C* y *la matriz de secuencia S*, junto con un registro de dirección de memoria *MAR* de *n* bits y un decodificador binario. El decodificador acepta como entrada la salida del registro *MAR* y selecciona una única línea de las 2^n posibles.

La matriz *C* consiste en un conjunto de líneas horizontales (las 2^n salidas del decodificador) y otro conjunto de líneas verticales. Los puntos marcados en la matriz *C* representan los acoplos de impedancia entre los dos conjuntos de líneas, esto es, cualquier señal eléctrica que se propaga a lo largo de una línea horizontal se acoplará para su propagación a través de las líneas verticales en aquellos puntos en los que exista una impedancia de acoplo.

Cada línea vertical de la matriz *C* está conectada a un único punto de control de la unidad de procesamiento y se corresponde por lo tanto con una microorden específica que se ejecuta cuando se activa la línea de control asociada en la matriz *C*. Las líneas horizontales de la matriz de control *C*, que contienen un conjunto de microórdenes, representan las microinstruciones.

Las señales eléctricas que se propagan por las líneas horizontales llegan también hasta la matriz de secuencia *S*. Las líneas verticales de salida de esta matriz constituyen la dirección de la siguiente microinstrucción que se almacenará en el registro *MAR*.

La capacidad de toma de decisiones por parte de la unidad de control con la subsiguiente ruptura de secuencia, la proporciona una señal de condición enviada a la unidad de control desde la unidad de procesamiento.

Se puede resumir esta exposición del modelo original de Wilkes en los puntos siguientes:

1) El número de líneas horizontales que emanan del decodificador representan el número de microinstrucciones que se pueden almacenar.
2) El número total de líneas verticales que tienen las matrices *C* y *S* define la longitud de palabra de la memoria de control.
3) Las líneas verticales de la matriz *C* (microórdenes) no necesitan decodificarse ya que son asignadas directamente a los puntos de control en una correspondencia uno a uno.
4) La inserción de las señales de condición tiene lugar entre la matriz de control *C* y la de secuencia *S*.

A partir de este planteamiento inicial se han propuesto numerosas modificaciones. A continuación se exponen los conceptos básicos de la microprogramación y las variaciones más interesantes a la propuesta de Wilkes.

Sección 7-2

Estructura de una unidad de control microprogramada

7.2.1 Conceptos básicos

Las dos tareas básicas realizadas por una unidad de control microprogramada son:

1) *Secuenciamiento de las microinstrucciones* es decir, obtener la próxima microinstrucción de la memoria de control
2) *Ejecución de la microinstrucción,* consiste en generar las señales de control que se necesitan para ejecutar la microinstrucción

Una microinstrucción es un conjunto de microórdenes que se pueden ejecutar de forma simultánea y que está contenida en una palabra de la memoria de control. Cada microinstrucción estaría representada por un conjunto diferente de 1's y 0's contenidos en las palabras de control.

7.2.2 Elementos de una unidad de control microprogramada

Se pueden distinguir dos bloques principales en la unidad de control de un procesador microprogramado (ver Figura 7.2): uno genera las microórdenes (la memoria de control) y el otro determina el paso siguiente en la secuencia de ejecución (el secuenciador):

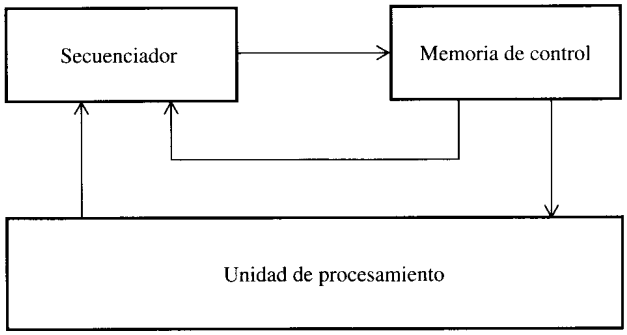


Figura 7.2: Elementos de una unidad de control microprogramada

1) *Memoria de control.* La memoria de control o micromemoria contiene toda la información necesaria para realizar el control del procesador. Una parte de esta información se envía al secuenciador para calcular la dirección de la siguiente microinstrucción del microprograma

$d(\mu I(i))$, y la otra parte que contiene las m microórdenes $\mu_{ord1}(i)$, $\mu_{ord2}(i)$, ..., $\mu_{ordm}(i)$, se envía a la unidad de procesamiento para realizar las microoperaciones del procesador. Posteriormente se considerará la organización de la información en el interior de la micromemoria.

2) *Secuenciador.* El secuenciador contiene un registro que guarda la dirección de la microinstrucción presente, y unos circuitos lógicos que calculan la dirección de la microinstrucción futura. El campo $d(\mu I(i))$ de la microinstrucción (7.2) está compuesto de microórdenes que controlan las transferencias de información entre el *registro de dirección de la memoria de control* (*RDC*) y las diferentes fuentes de la dirección futura del microprograma. La arquitectura del secuenciador determina las estructuras de control que se pueden emplear para la construcción de los microprogramas, es decir los tipos de secuencias que podrán seguir las microinstrucciones.

En la Figura 7.3 se muestran los elementos claves de una unidad de control microprogramada. El *registro de dirección de la memoria de control* (*RDC*) mantiene la dirección de la próxima microinstrucción que se va a leer. Cuando se lee una microinstrucción de la memoria de control, se transfiere al *registro de datos de la memoria de control* (*RMC*). El contenido de este registro son las señales de control que genera la unidad de control por lo que leer una microinstrucción de la memoria de control es lo mismo que ejecutarla. El tercer elemento de la Figura 7.3 es el secuenciador que carga el *RDC* y emite una orden de lectura.

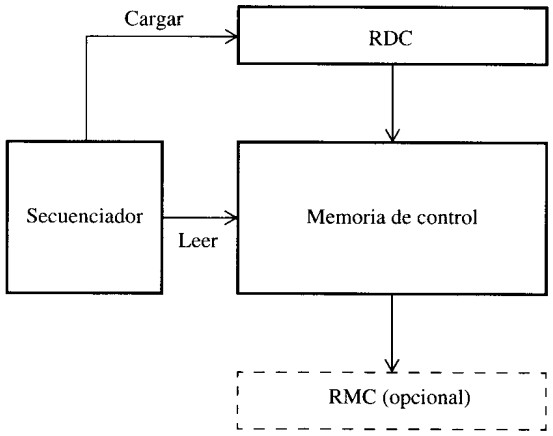


Figura 7.3: Arquitectura básica de una Unidad de Control microprogramada

En la Figura 7.4 se muestra con un mayor grado de detalle el funcionamiento de una unidad de control microprogramada que consta de los siguientes pasos (todos ellos suceden en un intervalo de reloj):

1) Para ejecutar una microinstrucción, el secuenciador emite una orden de lectura a la memoria de control.

2) La palabra cuya dirección se especifica en *RDC* se lee en *RMC*

3) El contenido de *RMC* genera las señales de control e información sobre la dirección de la próxima microinstrucción que envía al secuenciador.

4) El secuenciador carga una nueva dirección en *RDC*. Esta dirección se basa en la información

transmitida desde *RMC* sobre la próxima microinstrucción y en las señales de condición que la unidad de control microprogramada recibe de la unidad de procesamiento. Así pues las señales de condición afectan a la determinación del próximo estado de la unidad de control.

A menos que las señales de condición se salten la unidad de control y controlen directamente las microoperaciones que se ejecutan en la unidad de procesamiento, no pueden hacer nada más que influir sobre la próxima microoperación modificando la dirección que genera el secuenciador.

Este hecho sugiere de forma natural la utilización de un modelo de máquina secuencial tipo Moore para especificar el funcionamiento de una unidad de control microprogramada. Como consecuencia de ello las salidas en cualquier instante de tiempo son únicamente función del estado en que se encuentra la máquina secuencial en ese momento.

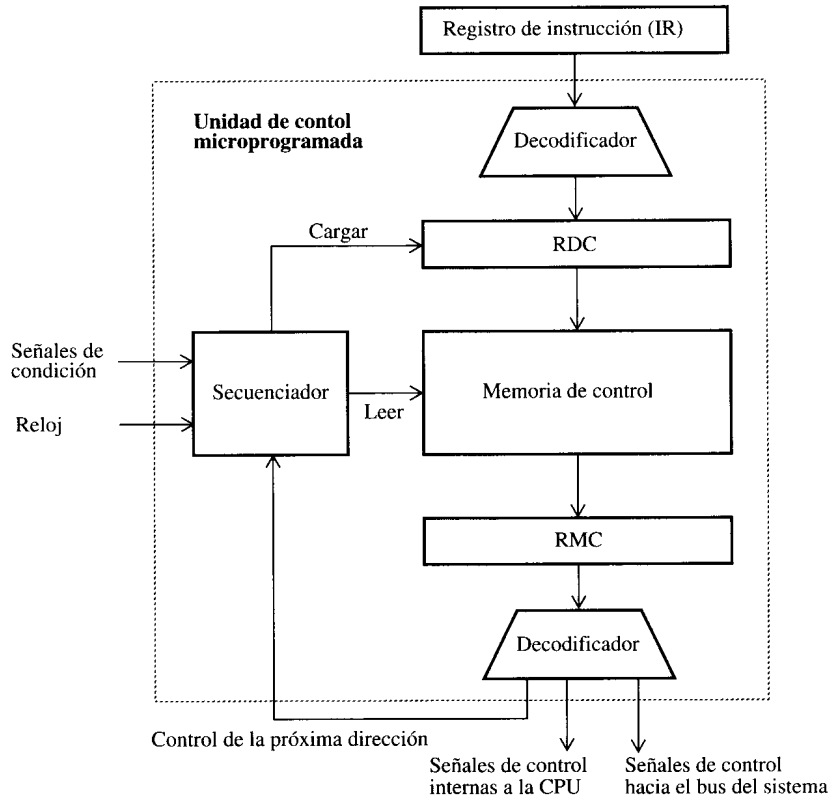


Figura 7.4: Funcionamiento de una unidad de control microprogramada

El paso 4 se puede elaborar algo más. A la conclusión de cada microinstrucción, el secuenciador carga una nueva dirección en *RDC*. Dependiendo del valor de las señales de condición externas y del contenido de *RMC* se realiza una de las tres acciones siguientes:

a) Se obtiene la dirección de la próxima microinstrucción sumando 1 al contenido de *RDC* ($RDC \leftarrow RDC + 1$). Es la más usual.

b) Se bifurca a una nueva secuencia de microinstrucciones mediante una microinstrucción de salto, cargando el campo de dirección de *RMC* en *RDC* (*RDC* ← *RMC*[*Dirección*]).

c) Se bifurca al comienzo del microprograma específico de ejecución de la siguiente instrucción máquina cargando el *RDC* con una dirección que es función del código de operación almacenado en el registro de instrucción (*RDC* ← *Función* (*IR*[*cod_op*])). Este tipo de salto, que ocurre sólo una vez por ciclo de instrucción, está determinado por el contenido de *IR* y se realiza al comienzo del ciclo de ejecución de la instrucción.

El registro *RMC*, no obstante es opcional y en determinadas arquitecturas de unidades de control microprogramadas las microinstrucciones se toman directamente de las salidas de la memoria de control, que es una memoria tipo *ROM*. La memoria *ROM* opera como un circuito combinacional, que tiene como entrada la dirección dada por el registro *RDC* y como salida la microinstrucción correspondiente. El contenido de la palabra especificada en la *ROM* permanece en sus líneas de salida mientras la dirección esté presente a su entrada. No se necesita una señal de $\overline{lectura/escritura}$ como en una memoria *RAM*. Cada pulso de reloj ejecuta las microoperaciones contenidas en la microinstrucción y también transfiere una nueva dirección al registro *RDC* que en este caso es el único componente de la unidad de control que recibe los pulsos de reloj y almacena información del estado. El secuenciador y la memoria de control son circuitos combinacionales. *El estado de la unidad de control viene dado por el contenido del registro RDC.*

En los dos apartados siguientes se presentan de forma más detallada estos dos elementos: el secuenciamiento de las microinstrucciones y la organización de la memoria de control, y se analizan las distintas alternativas de diseño de una unidad de control microprogramada.

7.2.3 Secuenciamiento de las microinstrucciones

El secuenciador más simple está formado por un contador. A la microinstrucción presente le seguirá siempre aquella que está situada en la dirección actual incrementada en 1 (ver Figura 7.5). Es evidente que muy pocos procesos podrán ser microprogramados con un secuenciador con este tipo de direccionamiento implícito. En la mayor parte de los casos el procesador deberá tomar una decisión y escoger una secuencia entre varias posibles, por ejemplo en función del resultado de una microoperación o de alguna señal de condición generada en la unidad de procesamiento.

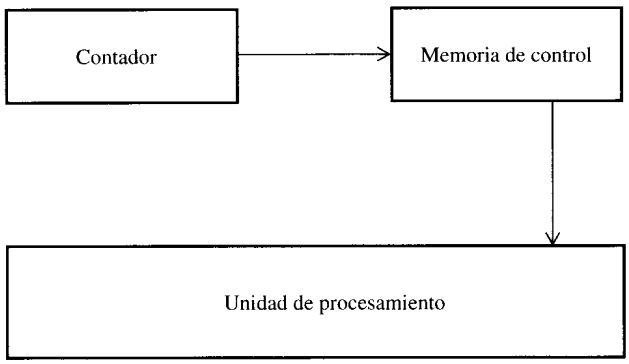


Figura 7.5: Contador como secuenciador de la memoria de control

De esta manera, al secuenciador elemental de la Figura 7.5 será necesario añadirle un dispositivo que permita elegir entre varias direcciones. Si esta elección es binaria, la propia microinstrucción puede contener las dos direcciones de manera explícita, o lo que es más usual proporcionar solamente una ya que la otra viene dada de forma implícita incrementando la dirección actual (ver Figura 7.6).

A pesar de que estas dos estructuras de control, el *salto condicional* y la *incrementacion de la dirección actual* son suficientes para microprogramar cualquier proceso, la claridad del microprograma y/o su facilidad de elaboración y de depuración demandan otras estructuras, otras maneras de secuenciar las microinstrucciones. Entre las más conocidas de estas estructuras se pueden citar: el salto a un subprograma, el retorno de un subprograma, el salto a la direccion inicial, la iteración, etc. En general, estas nuevas estructuras de control añaden al secuenciador únicamente nuevas fuentes de cómo especificar la dirección de la siguiente microinstrucción; por ejemplo una pila, para la utilización de subprogramas.

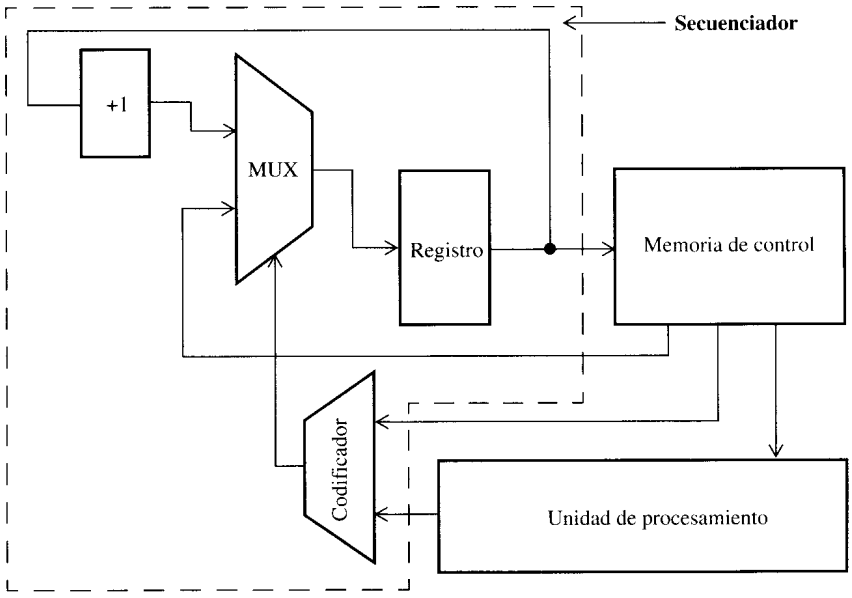


Figura 7.6: Secuenciador que permite el salto condicional entre microinstrucciones

La Figura 7.6 pone de manifiesto que para determinar la siguiente microinstrucción que se ha de ejecutar existen dos formas básicas de hacerlo: *direccionamiento explícito* y *direccionamiento implícito.*

Direccionamiento explícito

La dirección de la próxima instrucción se especifica en un campo separado tal como se indica en la Figura 7.7. Esta dirección se carga en el registro de dirección de la memoria de control (*RDC*). Cuando en el microprograma existen rupturas de secuencia condicional se necesitan dos o más direcciones por microinstrucción. En este esquema la memoria de control puede ser vista como una lista encadenada. El direccionamiento explícito es muy flexible y rápido, pero requiere microinstrucciones con longitudes de palabra grandes. En la Figura 7.8 se muestra la arquitectura de una unidad de control microprogramada con dos direcciones por microinstrucción.

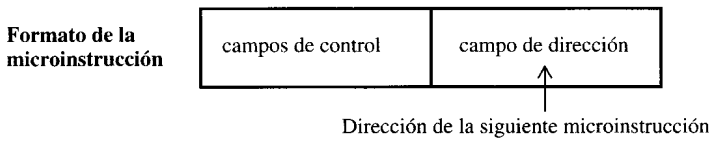


Figura 7.7: Direccionamiento explícito

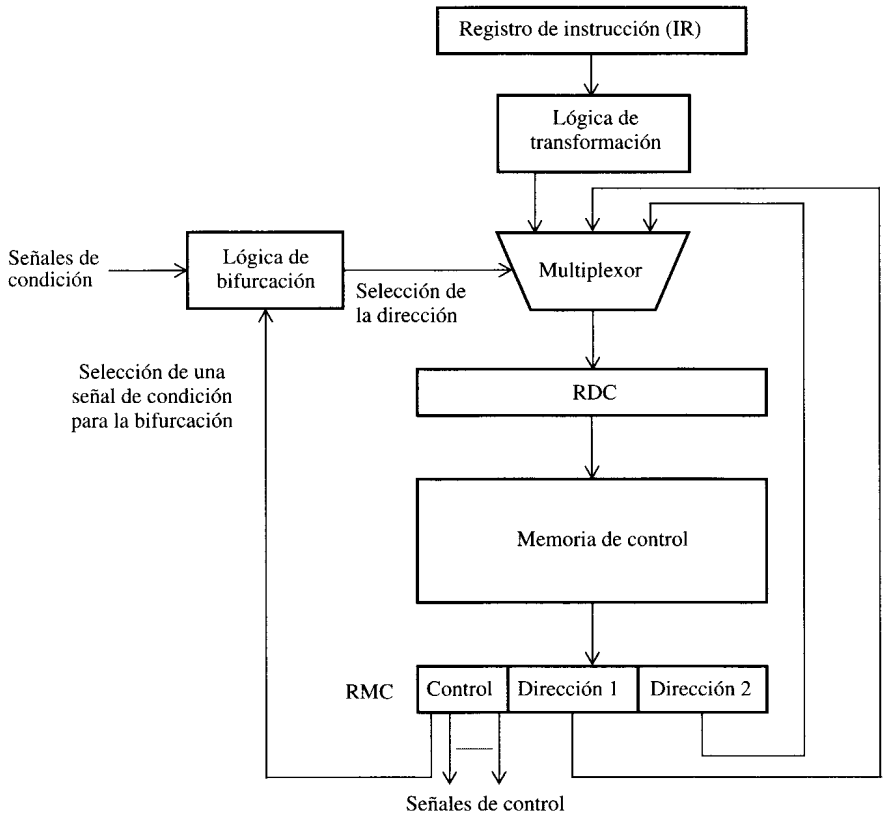


Figura 7.8: Unidad de control microprogramada con dos direcciones por microinstrucción

Cada instrucción tiene asociado su propio microprograma, almacenado a partir de una posición determinada de la memoria de control. La transformación de los bits del código de operación de la instrucción, en una dirección de la memoria de control donde está localizado el microprograma, la realiza el bloque denominado *lógica de transformación*. Un procedimiento de transformación es una regla que convierte el código de la instrucción en una dirección de la memoria de control.

El bloque *lógica de bifurcación* proporciona la capacidad de toma de decisiones en la unidad de control microprogramada. Recibe como entradas las señales de condición generadas en la unidad de procesamiento de la CPU (por ejemplo la señal de rebose del sumador de la ALU, el bit de signo del acumulador, etc.) y los

bits correspondientes del campo de control de la microinstrucción, que permiten seleccionar la condición cuyo valor se va a comprobar, y genera como salida la selección de la dirección de la próxima microinstrucción. Esta salida es la entrada de control del multiplexor, que tiene como entrada de datos los dos campos de dirección de la microinstrucción y la dirección asociada al código de la instrucción almacenada en el registro de instrucción. El formato con dos campos de dirección por microinstrucción produce, como se ha dicho, memorias de control con longitudes de palabras grandes.

Otra posibilidad de direccionamiento explícito que mejora este aspecto consiste en utilizar un formato de microinstrucción con sólo un campo de dirección (ver Figura 7.9). Con este esquema, las opciones para la selección de la siguiente dirección son:

- El campo de dirección de la microinstrucción
- La dirección asociada con el código de operación de la instrucción almacenada en el registro *IR*
- La siguiente dirección en secuencia

La señal de selección de la dirección determina qué opción se escoge. Se observa que, si la selección de la dirección es la de la siguiente en secuencia no se utiliza el campo de dirección de la microinstrucción, lo que supone también una cierta ineficacia en la forma de codificarla. El direccionamiento ímplícito trata de optimizar la utilización de la memoria de control.

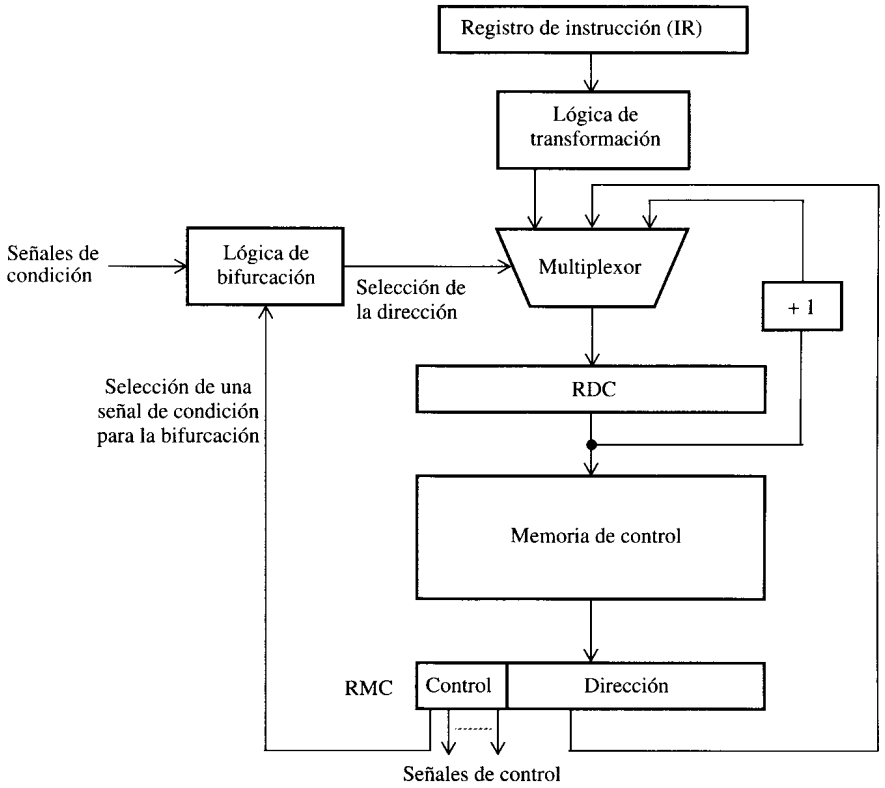


Figura 7.9: Unidad de control microprogramada con una dirección por microinstrucción

Direccionamiento implícito

El direccionamiento implícito permite reducir el tamaño de las microinstrucciones. Las microinstrucciones se ejecutan en el orden en que se almacenan en la memoria de control. Por lo tanto, no se necesita ninguna información de secuenciamiento en la propia microinstrucción que sólo contiene los campos de control. Esta solución es satisfactoria únicamente cuando el algoritmo a realizar no contiene ninguna ruptura de secuencia condicional.

Si existen estos tipos de bifurcaciones (que es lo normal) se precisa una microinstrucción especial para especificar la bifurcación. Así pues, el direccionamiento implícito requiere dos clases de microinstrucciones: una para especificar las señales de control y otra para especificar una bifurcación. En el formato de la microinstrucción de bifurcación, hay dos campos: el primero define la condición que se utiliza para efectuar la bifurcación y el segundo la dirección de ésta (ver Figura 7.10).

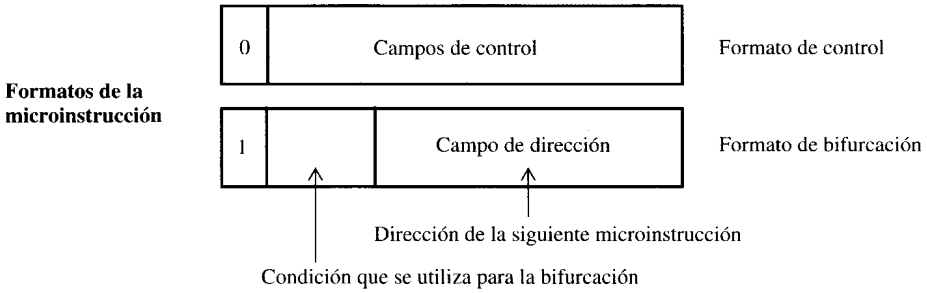


Figura 7.10: Direccionamiento implícito

Para calcular la dirección en el secuenciamiento implícito se hace lo siguiente:

1) Si la microinstrucción actual es del tipo de control o si es del tipo de bifurcación y no se satisface la condición de salto se incrementa en una unidad el contenido del registro de dirección de la memoria de control *RDC* (*RDC* ← *RDC* + 1).

2) Si se cumple la condición de salto se carga el registro *RDC* con la dirección de bifurcación.

Cuando se le compara con el esquema de secuenciamiento explícito presenta la ventaja de que requiere longitudes de palabras más cortas, aunque los microprogramas tienden a ser más largos, ya que por cada bifurcación se añade una nueva microinstrucción. En la Figura 7.11 se muestra la arquitectura de una unidad de control que utiliza un formato de microinstrucción con direccionamiento implícito.

El bloque denominado *lógica de selección* activa las señales de control si el bit de selección de formato corresponde al de una microinstrucción de control. Con el formato de bifurcación se pueden realizar tanto rupturas incondicionales como condicionales. Una desventaja de la arquitectura de unidad de control microprogramada con direccionamiento implícito es que se consume un ciclo completo con cada microinstrucción de bifurcación.

Con los direccionamientos explícitos la generación de la dirección ocurre durante el mismo ciclo que la generación de las señales de control, lo que minimiza los accesos a la memoria de control. Por este motivo el direccionamiento implícito es preferible si las bifurcaciones son poco frecuentes en el programa.

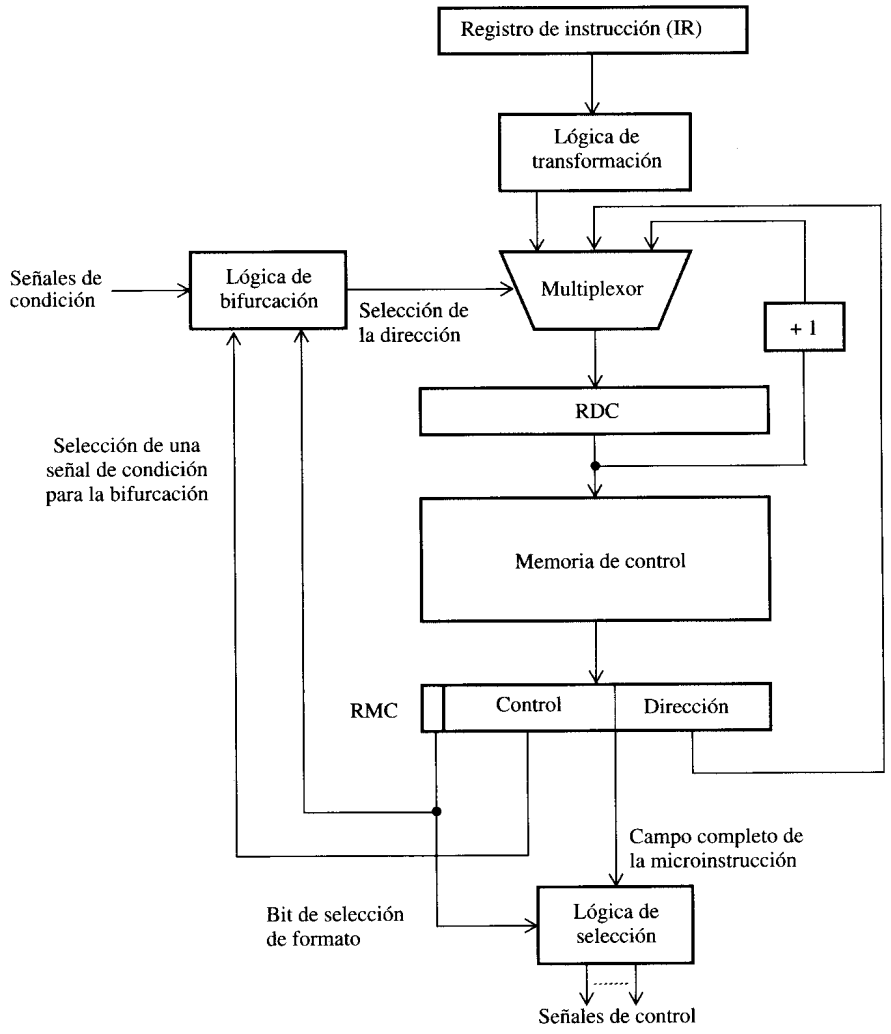


Figura 7.11: Unidad de control microprogramada con formato de microinstrucción variable

7.2.4 Organización de la memoria de control

En su formato más sencillo, una microinstrucción tiene un bit por cada señal de control y tiene una duración de un periodo. Esto permite la utilización en paralelo de los elementos del computador. Pero tiene como desventaja que las microinstrucciones son largas y están llenas de bastantes ceros (microórdenes desactivadas). Esto se debe a que muchas de las microórdenes son incompatibles entre sí porque las señales de control asociadas no pueden operar de forma simultánea ya que actúan sobre los mismos recursos de cálculo. Para reducir su longitud, las microinstrucciones se codifican, lo que reduce el grado de paralelismo y requiere una etapa de decodificación adicional.

Como ya se ha visto, cada microinstrucción, es decir cada palabra de la memoria de control, se puede descomponer en dos partes: una que envía microórdenes a la unidad de procesamiento y otra que envía microórdenes al secuenciador. Así pues, se puede considerar que una microinstrucción está compuesta de dos *campos*: *campo de control* y *campo de dirección*. Estos dos campos definen por una parte las señales de control que gobiernan la unidad de procesamiento o ruta de datos y por otra las señales de control que generan la dirección de la próxima microinstrucción en el secuenciador. El campo de control que especifica los valores de las señales de control durante la ejecución de una microinstrucción está dividido en *subcampos*, cada uno de los cuales gobierna una microoperación determinada. Por lo tanto el número de microoperaciones concurrentes no puede ser mayor que el número de subcampos de control.

Las microórdenes enviadas al secuenciador, su cantidad y su funcion, están directamente relacionadas con las estructuras de control escogidas para el microprograma. Existen, no obstante, diferentes estrategias para organizar las microórdenes que controlan la unidad de procesamiento. El formato de las microinstrucciones no codificadas recibe el nombre de *horizontal*, mientras que el de las codificadas recibe el nombre de *vertical*. Dependiendo del grado de codificación existen soluciones intermedias entre ambos extremos.

1) *Microprogramación horizontal.* Una primera solución consiste en conectar cada bit de la micromemoria con un punto físico de control de los diferentes recursos de la unidad de procesamiento (ver Figura 7.12). Cada bit de la memoria de control, que es un subcampo independiente, corresponde a una sola microorden. Esta manera de trabajar permite la realización de la mayor cantidad posible de microórdenes por microinstrucción, puesto que todas pueden ser controladas de forma simultánea. El formato horizontal proporciona así la máxima concurrencia posible de microórdenes. Las señales de control están ya decodificadas y su generación es muy rápida, sin embargo el precio que se paga es una memoria demasiado ancha si el número de microórdenes es elevado.

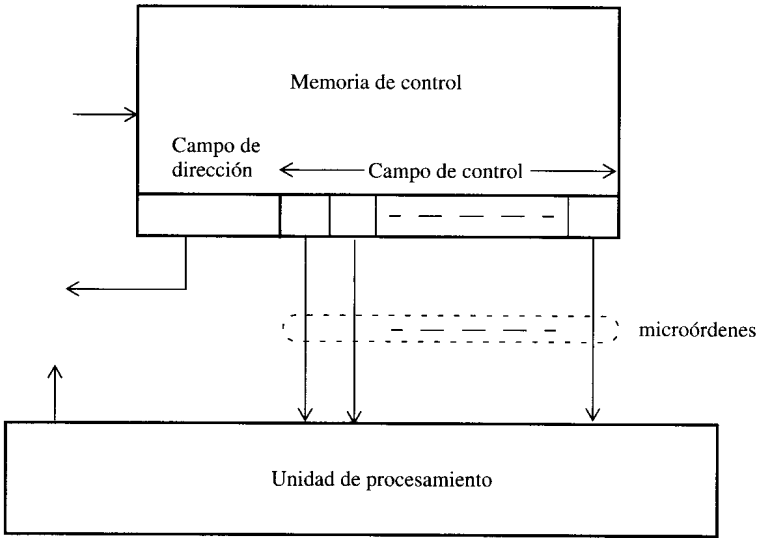


Figura 7.12: Formato horizontal de una microinstrucción

2) *Microprogramación vertical.* La microprogramación puramente horizontal es, en la mayor parte de los casos, redundante ya que en la práctica existe siempre un número limitado de microoperaciones que se pueden realizar en paralelo. Por ejemplo, en un momento dado sólo se permite transferir el contenido de un único registro fuente a un bus de datos. Es entonces posible codificar esta información y en lugar de transmitirr los n bits que controlan directamente las n fuentes, se puede utilizar con un decodificador adecuado un código con $log_2 n$ bits. El campo de control de la microinstrucción se descompone de esta manera, en varios subcampos, cada uno de los cuales controla un conjunto de operadores, de operandos o de caminos de datos (ver Figura 7.13). El formato vertical es así más lento que el formato horizontal, debido al tiempo que se necesita para la decodificación de los subcampos, pero tiene como ventaja que utiliza longitudes de palabras más reducidas y por lo tanto requiere memorias de control más pequeñas. Con un subcampo de j bits se pueden especificar 2^j-1 señales de control (una de las codificaciones hay que reservarla para el caso que ninguna señal de control esté activa). Como en cada momento sólo puede haber una señal activa por subcampo, hay que elegirlos de forma que se garantice que las operaciones que controlan nunca se necesitan de forma simultánea.

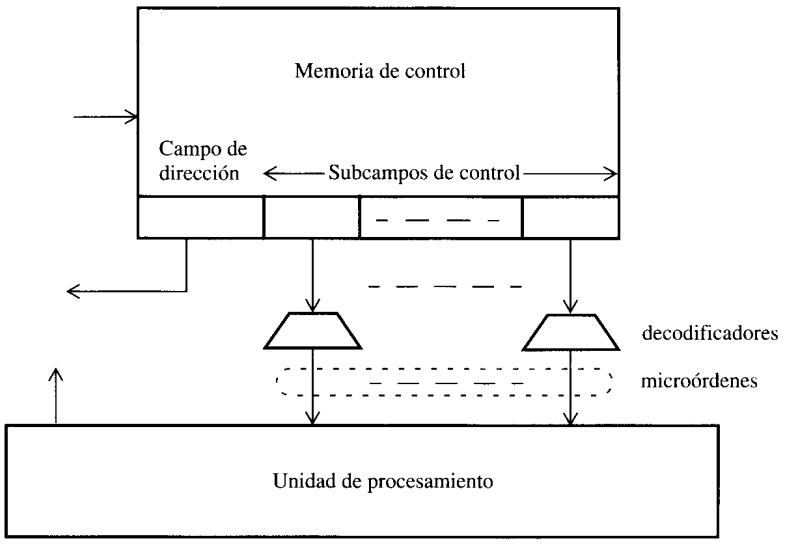


Figura 7.13: Formato vertical de una microinstrucción codificada por subcampos

La codificación de la microinstrucción puede llevarse a cabo hasta su límite superior; para hacerlo se efectúa un análisis exhaustivo del sistema para determinar las n combinaciones posibles o deseadas de las microoperaciones y se le da un código diferente a cada una de ellas (ver Figura 7.14). El ancho de la micromemoria se reduce aun más, pues sólo es necesario un decodificador, pero el sistema pierde cierta flexibilidad de utilización.

La microprogramacion vertical también efectúa otra codificación de la microinstrucción: en este caso *cada bit de la memoria de control no tiene siempre la misma función*. En efecto, si dos conjuntos de operandos, de operadores o de caminos de datos no se emplean nunca al mismo tiempo, los dos campos de control respectivos de la microinstrucción pueden ocupar la misma

posición. Sólo será necesaria una indicación adicional para diferenciar las distintas funciones de los campos y los diferentes formatos de la microinstrucción. El ancho de la memoria de control final se reduce aun más. Este nuevo campo se denomina algunas veces *campo de selección* (*SEL*), pues permite seleccionar el código del formato de la microinstrucción (ver Figura 7.15).

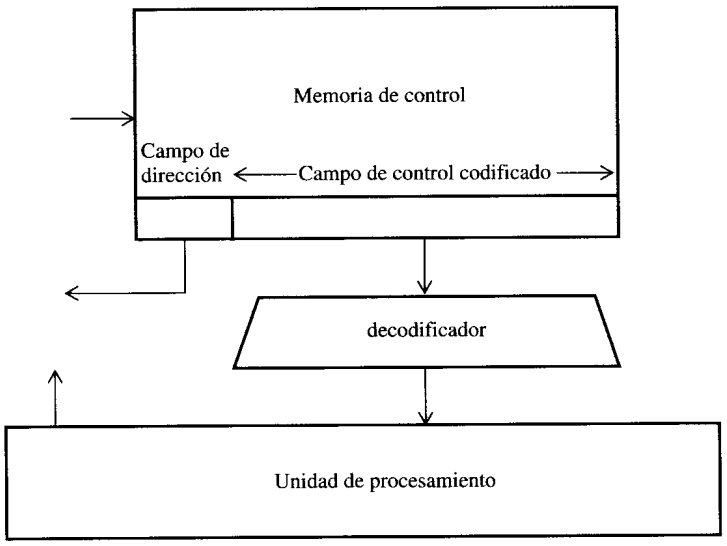


Figura 7.14: Formato vertical de una microinstrucción totalmente codificada

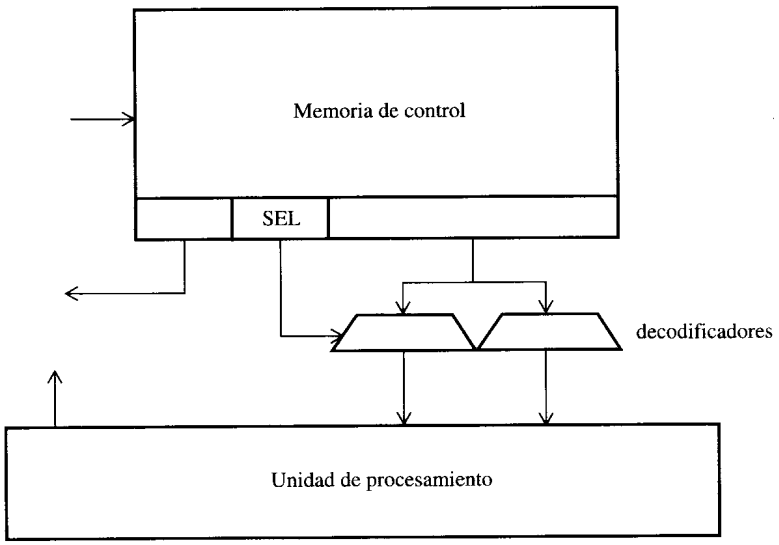


Figura 7.15: Formato vertical con campo de selección del tipo de microinstrucción

Entre los dos casos extremos de microprogramación puramente horizontal o vertical existen otros casos, mezclas de los dos, que no se pueden clasificar fácilmente en una de esas dos categorias. En conclusión se puede decir que la microprogramación puramente horizontal realiza un enfoque estrictamente hardware de la microprogramación (cada bit de la micromemoria está conectado directamente a un punto físico de la unidad de procesamiento) mientras que la microprogramación enteramente codificada corresponde a un planteamiento más cercano a la programación (se establece un repertorio de las microoperaciones que van a utilizarse).

En algunos casos las microoperaciones que componen una microinstrucción no han de activarse simultáneamente, sino que han de hacerlo consecutivamente a lo largo de más de un subciclo de reloj. En este caso, la unidad de control dispondrá de un *generador de subciclos* o *contador de periodos*. De este modo se pueden conseguir microinstrucciones más compactas, garantizando a la vez el correcto secuenciamiento de las microoperaciones. Las instrucciones de un programa se ejecutan de manera secuencial, salvo cuando se ejecuta una instrucción de bifurcación. Existen ocasiones, no programadas, en las que se debe suspender la ejecución de un programa para pasar a ejecutar otro. Cuando la causa de esta ruptura de secuencia es interna a la CPU (por ejemplo error de paridad en la memoria, error en la ejecución de una instrucción por desbordamiento, división por cero, etc.) se denomina *trap*. Si la causa es externa (por ejemplo originada por un periférico) se habla de *interrupción* (ver sección 3-4).

7.2.5 Ejecución de las microinstrucciones

De la misma forma que en la ejecución de una instrucción máquina, la ejecución de una microinstrucción requiere una serie de fases. El ciclo de ejecución de una microinstrucción consta de las siguientes fases:

1) Búsqueda (lectura) de una microinstrucción
2) Decodificación de los campos de la microinstrucción
3) Ejecución de las microoperaciones
4) Cálculo de la dirección de la siguiente microinstrucción

Las fases 3) y 4) se pueden realizar simultáneamente si las condiciones de bifurcación se determinan en la microinstrucción previa. La Figura 7.16 muestra un diagrama de temporización característico del ciclo de una microinstrucción.

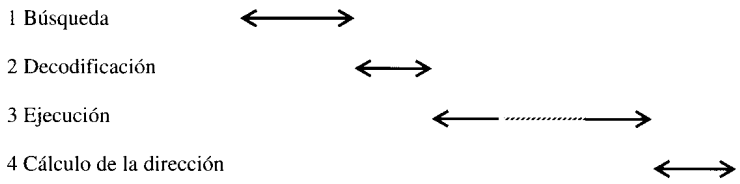


Figura 7.16: Ciclo de una microinstrucción

Atendiendo a la forma en que cada microinstrucción se ejecuta se pueden clasificar en *monofásicas* y *polifásicas*.

a) *Monofásicas*

Las microoperaciones especificadas en la microinstrucción se pueden ejecutar todas al mismo tiempo.

b) *Polifásicas*

Son aquellas microinstrucciones cuyos campos se utilizan de forma escalonada (en ciclos menores o subciclos consecutivos) a lo largo del tiempo de ejecución tal como se muestra en la Figura 7.17. Este esquema resulta eficaz cuando la búsqueda de las microinstrucciones es lenta en comparación con su ejecución.

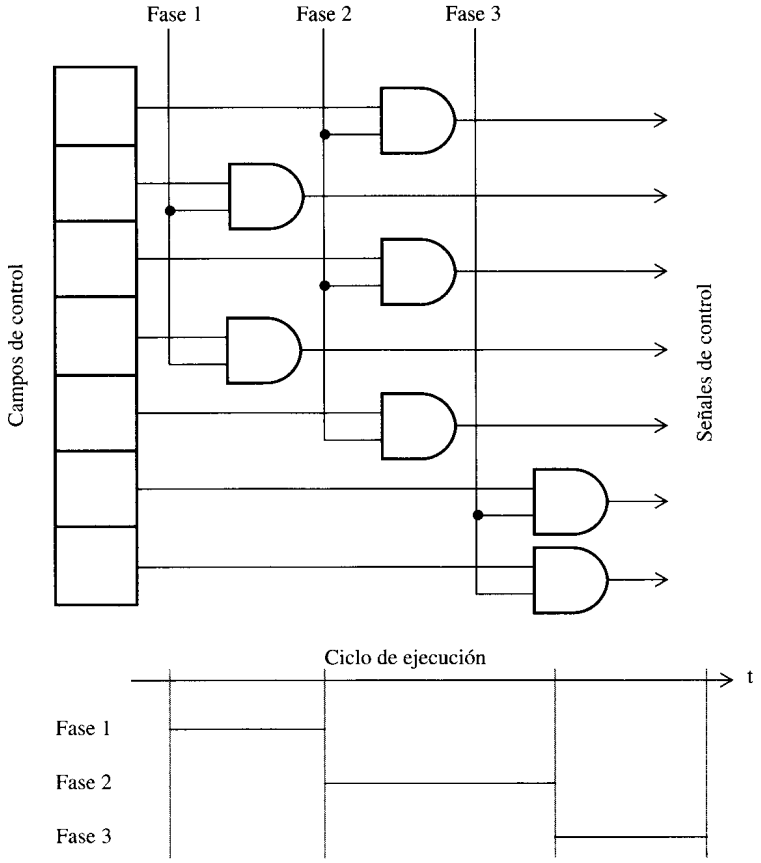


Figura 7.17: Microinstrucción polifásica

Sección 7-3

Representación de los microprogramas

Un microprograma se puede representar utilizando un grafo orientado (ver Figura 7.18). En el interior de cada nodo aparecen las microórdenes enviadas por cada microinstrucción a la unidad de procesamiento y las líneas indican la secuencia que debe seguirse, es decir el orden de ejecución de las microinstrucciones. Cada línea está marcada con las condiciones que controlan la transición de una microinstrucción a la siguiente.

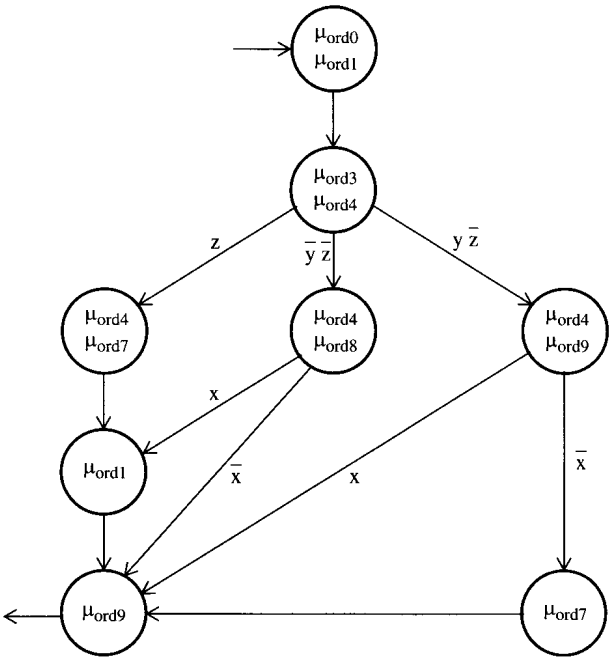


Figura 7.18: Representación de un microprograma como un grafo orientado

Este parecido con los diagramas de estados utilizados para representar las máquinas secuenciales no debe sorprender: en efecto ya se ha indicado que la unidad de control, intérprete del microprograma, es una máquina secuencial. Una microinstrucción es entonces equivalente a un estado de la máquina secuencial representada por el microprograma. En esta representación se ven claramente las dos funciones del microprograma (que corresponden a las dos partes del procesador).

1) Los nodos representan las microoperaciones ejecutadas por la unidad de procesamiento.

2) Las líneas representan las secuencias realizadas por la unidad de control.

Tal como se ha visto sólo son necesarias y suficientes dos estructuras de control en un microprograma: la *transición incondicional* (realizada generalmente con un incremento de la dirección del secuenciador) y la *transición condicional* (realizada con la elección de una entre varias direcciones en función de una o varias condiciones). Este hecho se refleja en el grafo orientado del microprograma de la manera siguiente: si la transición es incondicional, el nodo de la microinstrucción correspondiente tendrá una sóla línea de salida, sin ninguna marca; si la transición es condicional, el nodo de la microinstrucción correspondiente tendrá varias líneas de salida, una por cada microinstrucción futura posible, marcadas con las condiciones de dicha transición. Las condiciones de transición son expresiones lógicas de una o varias variables, en general funciones de los resultados de las microoperaciones. La expresión lógica debe ser evaluada por el secuenciador, para escoger la microoperación futura (operación equivalente a la del sistema lógico combinacional que calcula el próximo estado de una máquina secuencial, en función del estado presente y de las variables de entrada). Esta evaluación puede ser realizada en paralelo, todas las variables al mismo tiempo (equivalente a la instrucción *case* del lenguaje Modula 2), o en serie, variable por variable (equivalente a un conjunto de instrucciones *if...then...else* del lenguaje Modula 2). Este último caso conduce a otro modo de representación de los microprogramas: el diagrama de flujo u organigrama. La Figura 7.19 es, por ejemplo, un organigrama equivalente al grafo orientado de la Figura 7.18: los nodos se representan ahora por rectángulos, y las transiciones condicionales mediante rombos que contienen una serie de comprobaciones sobre las variables de condición.

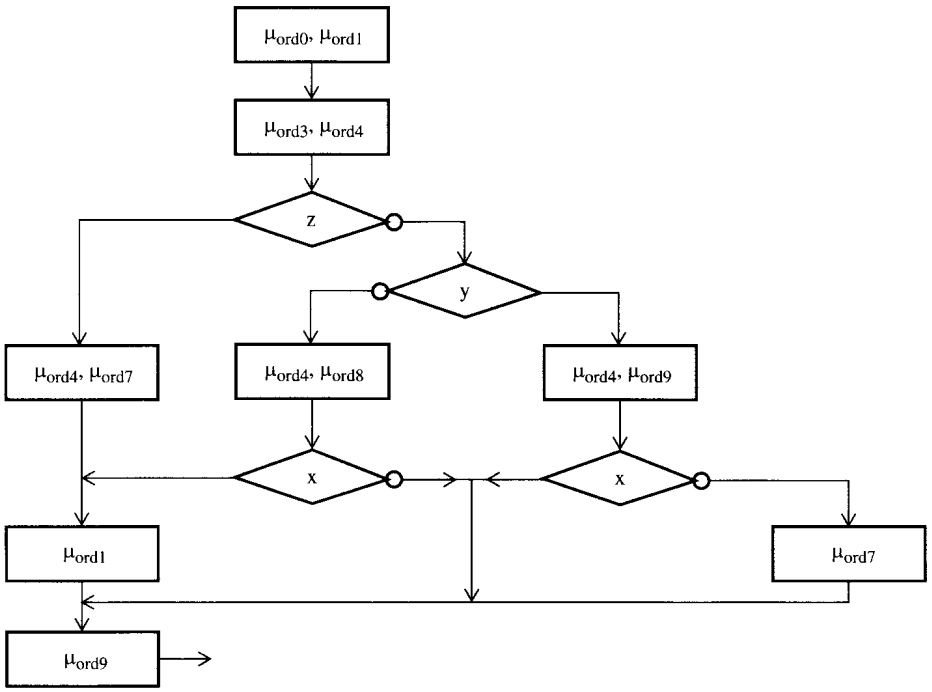


Figura 7.19: Representación de un microprograma como un organigrama o diagrama de flujo

La representacion del organigrama sugiere una microprogramación muy vertical: cada elemento del organigrama, ya sea rectángulo o rombo, será ahora una microinstrucción; la microinstrucción horizontal original se descompone de esta manera en su parte de ejecución (envío de las microórdenes a la unidad de procesamiento o ruta de datos) y en su parte de comprobación (elección de la microinstrucción futura). El control del secuenciador está completamente separado del control de la unidad de procesamiento. Los mismos bits que en un momento dado sirven para formar la dirección de la microinstrucción siguiente, en otro momento permiten controlar la ruta de datos. El campo de selección que identifica el formato de la microinstrucción sirve también para indicar el tipo de secuencia que se seguirá. La economía hecha en el ancho de memoria se paga con una mayor cantidad de microinstrucciones. Un procesador, que no disponga de una unidad aritmético-lógica, y ejecute este tipo de microprogramación vertical se denomina *máquina de decisión binaria.*

Lógicamente, la evaluación en serie de una expresión lógica puede producir varias soluciones equivalentes aunque distintas. Así, por ejemplo, la Figura 7.19 es un organigrama equivalente al de la Figura 7.20. Una máquina de decisión binaria que ejecutase el organigrama de la Figura 7.20 necesitaría un microprograma más largo, tanto en tiempo como en tamaño de memoria, que el organigrama de la Figura 7.19.

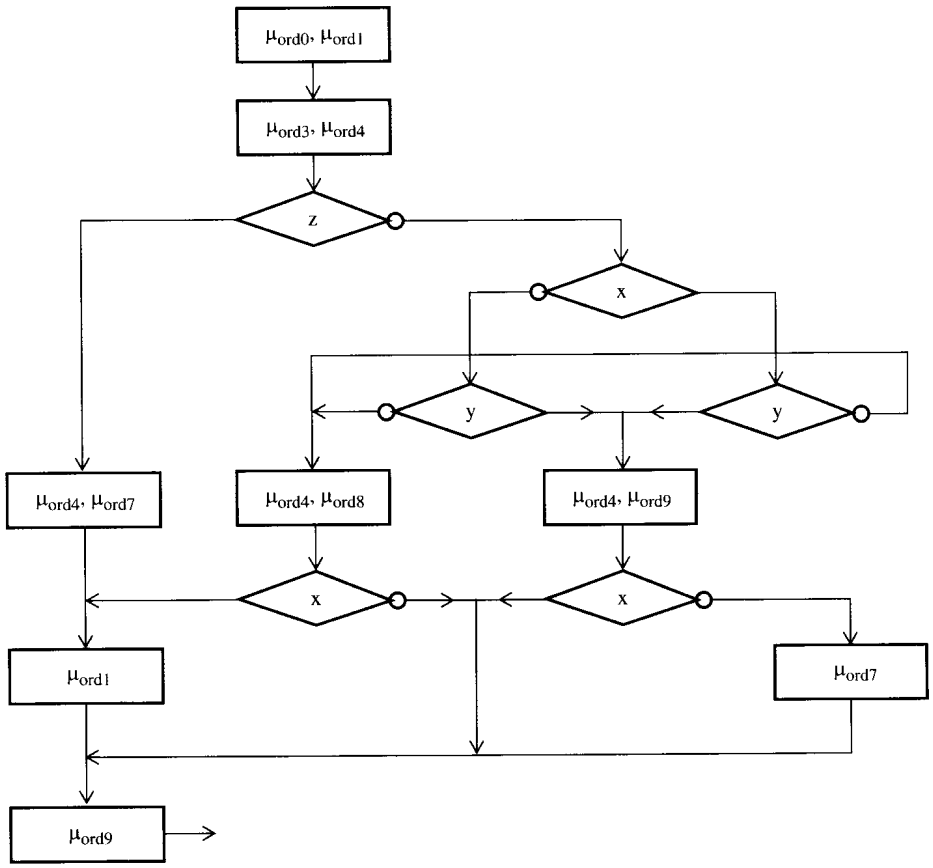


Figura 7.20: Organigrama equivalente al de la Figura 7.19

7.3.1 Ejemplo: Desarrollo de un sencillo microprograma

Con el fin de poner de manifiesto los conceptos introducidos en el apartado anterior se presentan de forma detallada las distintas alternativas que se pueden dar en el desarrollo de un sencillo microprograma. Sólo se especifican aquellos elementos de la ruta de datos y de la unidad de control del procesador que son necesarios para la realización del microprograma.

Especificaciones del procesador

Se dispone de un procesador microprogramable con una ruta de datos compuesta de los elementos siguientes (ver Figura 7.21):

M [*MAR*]	memoria *RAM* de 256 × 8 bits
T	registro de 8 bits
R_1	registro de 8 bits
MAR	registro de dirección de memoria de 8 bits
DECR (-1)	operador de decrementación
INCR (+1)	operador de incrementación
SUM	sumador

En la Tabla 7.1 se muestra el conjunto de microórdenes que es capaz de ejecutar este procesador en su ruta de datos.

Microorden	Significado	Acción
μ_{ord0}	Decrementa el registro T	T ← T − 1
μ_{ord1}	Transfiere T + 1 a MAR	MAR ← T + 1
μ_{ord2}	Transfiere T a MAR	MAR ← T
μ_{ord3}	Suma R_1 a M[MAR]	M[MAR] ← M[MAR] + R_1
μ_{ord4}	Transfiere M[MAR] a R_1	R_1 ← M[MAR]

Tabla 7.1: Microórdenes de la ruta de datos del procesador microprogramado

El secuenciamiento de las microoperaciones del procesador se especifica mediante las dos microórdenes que se dan en la Tabla 7.2. Sólo son posibles dos estructuras de control: la *secuencia normal* (salto incondicional a la microinstrucción siguiente) y el salto incondicional a la microinstrucción $\mu I(x)$. La dirección de salto *x* se especifica en el campo de dirección de la microinstrucción.

Microorden	Significado	Acción
μ_{ord5}	Secuencia normal	RDC ← RDC + 1
μ_{ord6}	Salto incondicional a μI(x)	RDC ← x

Tabla 7.2: Microórdenes para el secuenciamiento del procesador microprogramado

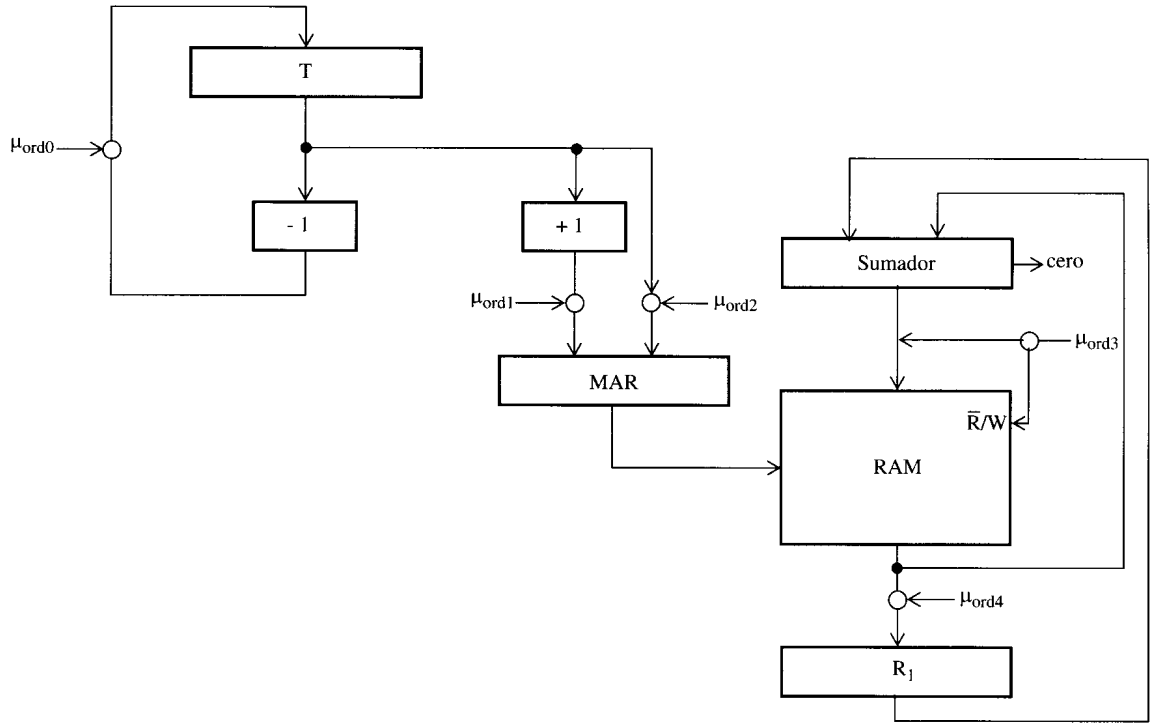


Figura 7.21: Ruta de datos del procesador del ejemplo 7.3.1

Objetivo del microprograma

Se pide microprogramar este procesador para realizar la función siguiente: decrementar el registro *T*, transferir luego a la posición de memoria dada por el valor de *T* el contenido presente de esa posición sumado al contenido de la posición de memoria siguiente; si el resultado de la suma es igual a 0, el microprograma debe bifurcar a la microinstrucción que se encuentra en la dirección *x*, en caso contrario continuará con la secuencia normal.

En otros téminos, el microprograma pedido debe realizar las acciones siguientes:

$$T \leftarrow T - 1 \tag{7.4}$$

$$M[T] \leftarrow M[T] + M[T+1] \tag{7.5}$$

$$\textbf{if } M(T) = 0 \textbf{ then } RDC \leftarrow x \textbf{ else } RDC \leftarrow RDC + 1 \tag{7.6}$$

donde *RDC* es el registro de dirección de la memoria de control .

En la Figura 7.22 se muestra un esquema de la unidad de control del procesador en el que se indica para el ejemplo propuesto la condición para que se produzca el salto a $\mu I(x)$ que depende de la variable de condición *cero* que proviene de la ruta de datos (indica que el resultado de la suma ha sido igual a 0). La ruptura de secuencia se produce si está activada la microorden μord_6 y *cero* = 1).

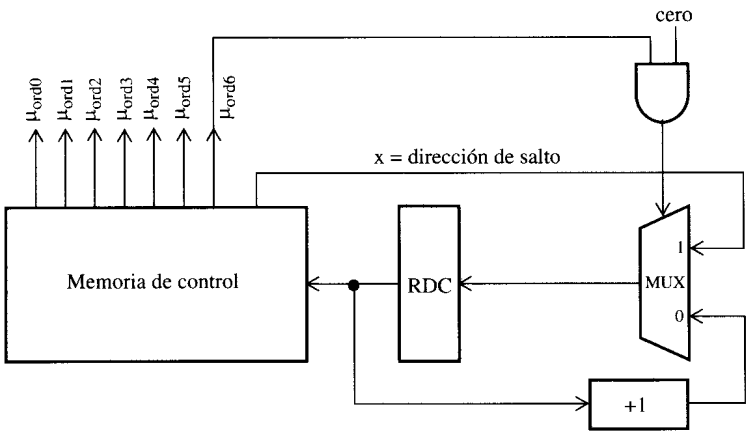


Figura 7.22: Unidad de control del procesador del ejemplo 7.3.1

Primera solución

Como las acciones (7.4), (7.5) y (7.6) no son todas microórdenes, el diseño del microprograma consistirá en encontrar una secuencia adecuada de las microórdenes: μ_{ord0}, μ_{ord1}, μ_{ord2}, μ_{ord3}, μ_{ord4}, μ_{ord5} y μ_{ord6} que genere un resultado equivalente. Una solución posible está dada por el grafo orientado de la Figura 7.23.

Si se admite una microprogramación completamente horizontal cada microorden estará controlada por un bit de la microinstrucción que especificará igualmente los bits de la dirección de salto. Es decir cada palabra de la memoria de control tendrá el formato de la Figura 7.24. Se reservan 8 bits para la dirección de salto del microprograma (porque la memoria contiene 256 palabras) y no se dispone de un campo que de manera explícita indique la variable de condicion.

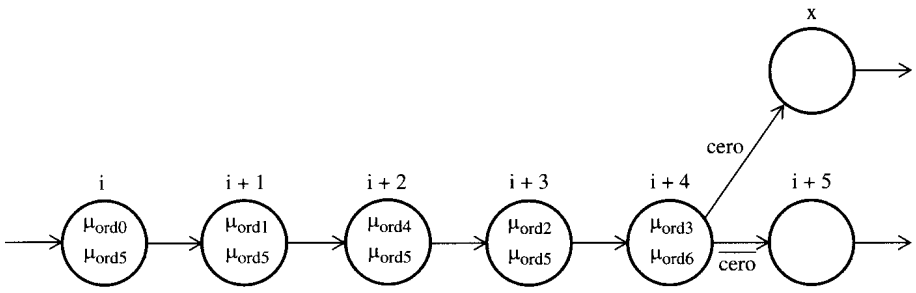


Figura 7.23: Grafo orientado del microprograma para la primera solución del ejemplo 7.3.1

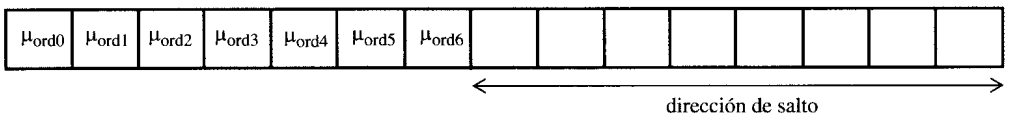


Figura 7.24: Formato de la microinstrucción para la primera solución del ejemplo 7.3.1

En la Tabla 7.3 se muestra el código binario del microprograma correspondiente a esta solución que está compuesto de 5 palabras de 15 bits.

Microprograma	Codificación binaria															
	μ_{ord0}	μ_{ord1}	μ_{ord2}	μ_{ord3}	μ_{ord4}	μ_{ord5}	μ_{ord6}	dirección de salto								
μI (i)	1	0	0	0	0	1	0	-	-	-	-	-	-	-	-	
μI (i + 1)	0	1	0	0	0	1	0	-	-	-	-	-	-	-	-	
μI (i + 2)	0	0	0	0	1	1	0	-	-	-	-	-	-	-	-	
μI (i + 3)	0	0	1	0	0	1	0	-	-	-	-	-	-	-	-	
μI (i + 4)	0	0	0	1	0	0	1	x								

Tabla 7.3: Código binario del microprograma para la primera solución del ejemplo 7.3.1

Segunda solución

El número de bits (15) de cada microinstrucción de la primera solución puede reducirse si las microórdenes se agrupan en campos y se codifican según su función. Por ejemplo, las dos microórdenes μ_{ord5} y μ_{ord6}, que controlan las dos estructuras de control del microprograma, pueden describirse con un sólo bit μ_0 ($\mu_0 = 0$ para la secuencia normal y $\mu_0 = 1$ para la bifurcación condicional).

Además, como en cada instante en la unidad de procesamiento sólo se ejecuta una única microorden de las cinco posibles (μ_{ord0} a μ_{ord4}), se pueden codificar mediante tres bits μ_1, μ_2 y μ_3 (ver Tabla 7.4).

μ_1	μ_2	μ_3	microorden
0	0	0	μ_{ord0}
0	0	1	μ_{ord1}
0	1	0	μ_{ord2}
0	1	1	μ_{ord3}
1	0	0	μ_{ord4}
1	0	1	-
1	1	0	-
1	1	1	-

Tabla 7.4: Codificación de las microórdenes μ_{ord0}, μ_{ord1}, μ_{ord2}, μ_{ord3} y μ_{ord4}

En la Figura 7.25 se muestra el nuevo formato de la microinstrucción y en la Tabla 7.5 el código binario del microprograma resultante (5 palabras de 12 bits).

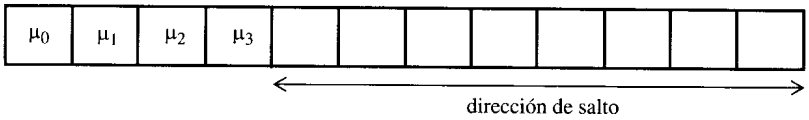


Figura 7.25: Formato de la microinstrucción para la segunda solución del ejemplo 7.3.1

Microprograma	Codificación binaria											
	μ_0	μ_1	μ_2	μ_3	dirección de salto							
μI (i)	0	0	0	0	-	-	-	-	-	-	-	-
μI (i + 1)	0	0	0	1	-	-	-	-	-	-	-	-
μI (i + 2)	0	1	0	0	-	-	-	-	-	-	-	-
μI (i + 3)	0	0	1	0	-	-	-	-	-	-	-	-
μI (i + 4)	1	0	1	1	x							

Tabla 7.5: Código binario del microprograma para la segunda solución del ejemplo 7.3.1

Esta reducción en el tamaño de la memoria de control va acompañada de una modificación de la unidad de control del procesador. Efectivamente cada campo de la microinstrucción debe ser decodificado antes de enviar las microórdenes. En la Figura 7.26 se muestra el nuevo esquema de la unidad de control. Si se compara con la de la Figura 7.22, puede observarse que hay un decodificador adicional.

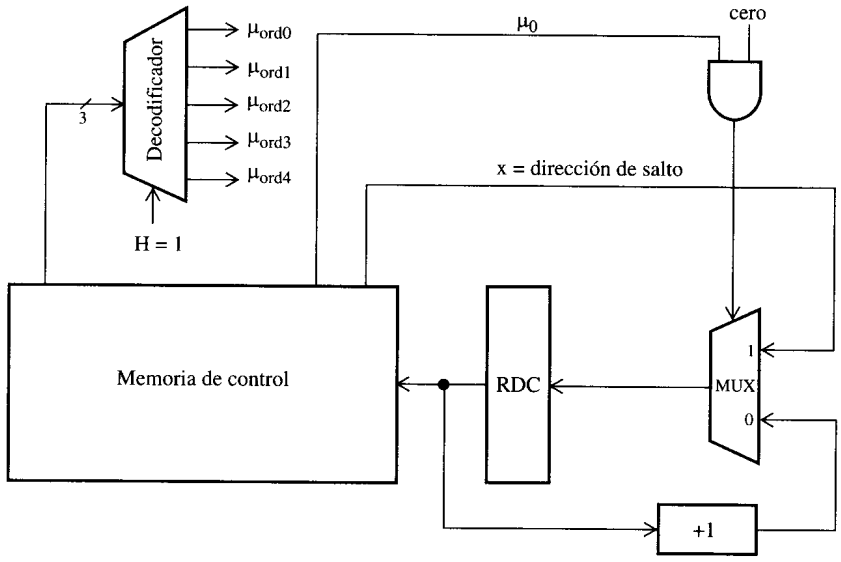


Figura 7.26: Nuevo esquema de la unidad de control del procesador del ejemplo 7.3.1

Tercera solución

Una nueva reducción del tamaño de la memoria de control se obtiene recurriendo a la microprogramación vertical. En efecto, en cuatro de las cinco microinstrucciones de las dos soluciones anteriores, los 8 bits de la dirección futura no tienen ningún significado: se podría pensar entonces en la utilización de dos formatos de microinstrucción, uno para el control de la unidad de procesamiento y otro para la transición condicional. Cada elemento rectángulo o rombo del organigrama de la Figura 7.27, equivalente al grafo orientado de la Figura 7.23, será una microinstrucción en uno u otro formato.

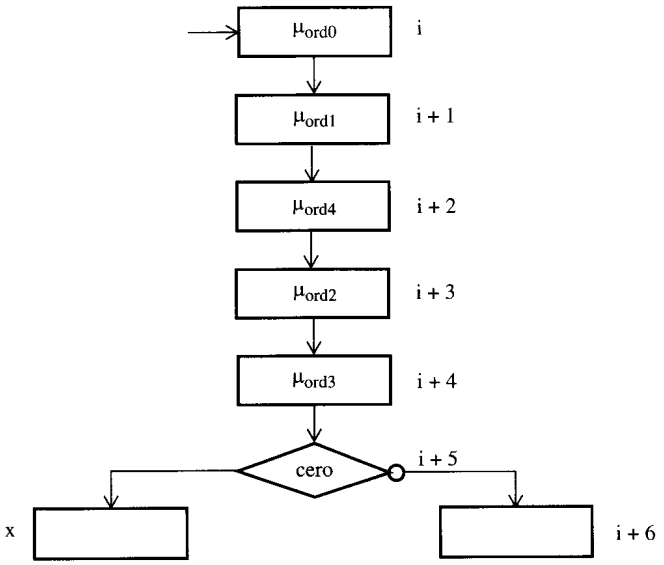


Figura 7.27: Organigrama equivalente al grafo orientado de la Figura 7.15

Un primer formato, el *formato de ejecución* que corresponde a los rectángulos del organigrama estará caracterizado por el valor 0 de un bit *OP* de la microinstrucción (equivalente al bit μ_0 de la solución anterior pero que aquí sirve para diferenciar los dos formatos); el valor 1 de este mismo bit va a caracterizar el *formato de test*, que corresponde a los rombos del organigrama. En la Figura 7.28 se muestran los dos formatos de microinstrucción para esta tercera solución del ejemplo 7.3.1.

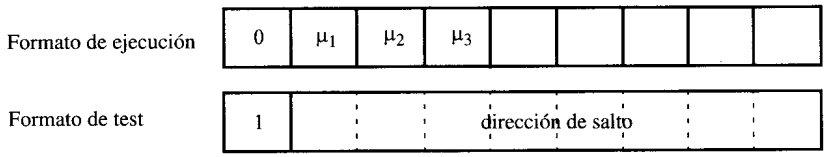


Figura 7.28: Formato de la microinstrucción para la segunda solución del ejemplo 7.3.1

El microprograma resultante tendrá una microinstrucción más, seis en lugar de cinco, pero como contrapartida sólo tienen nueve bits de anchura. En la Tabla 7.6 se muestra el código binario del microprograma resultante.

Microprograma	Codificación binaria								
	OP	μ_1	μ_2	μ_3					
μI (i)	0	0	0	0	-	-	-	-	-
μI (i + 1)	0	0	0	1	-	-	-	-	-
μI (i + 2)	0	1	0	0	-	-	-	-	-
μI (i + 3)	0	0	1	0					
μI (i + 4)	0	0	1	1	-	-	-	-	-
μI (i + 4)	1				x				

Tabla 7.6: Código binario del microprograma para la segunda solución del ejemplo 7.3.1

El precio de esta nueva reducción es una modificación de la unidad de control del procesador. Ahora además del decodificador de microórdenes es necesario inhibir todas las microórdenes en el momento de la ejecución de una microinstrucción de test. Esta desactivación de las microórdenes se realiza enviando a la unidad de procesamiento las microórdenes correspondientes a una operación neutra que se denomina *NOP* (no-operación). Cuando *OP* = 1 (identifica el formato de test) la línea de habilitación *H* del decodificador vale 0 y por consiguiente todas sus salidas (que son las microórdenes) se ponen a 0, lo que corresponde a la instrucción *NOP* (ver Figura 7.29).

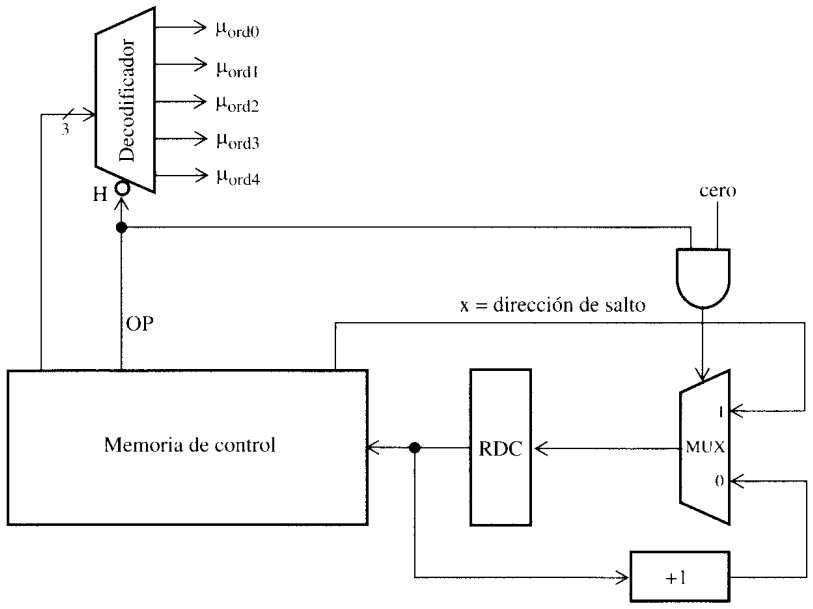


Figura 7.29: Esquema de la unidad de control para la tercera solución del ejemplo 7.3.1

Cuarta solución

Hasta aquí la reducción del tamaño del microprograma ha sido como consecuencia de una reorganización de la memoria de control y no de un cambio del propio microprograma o lo que es equivalente del algoritmo.

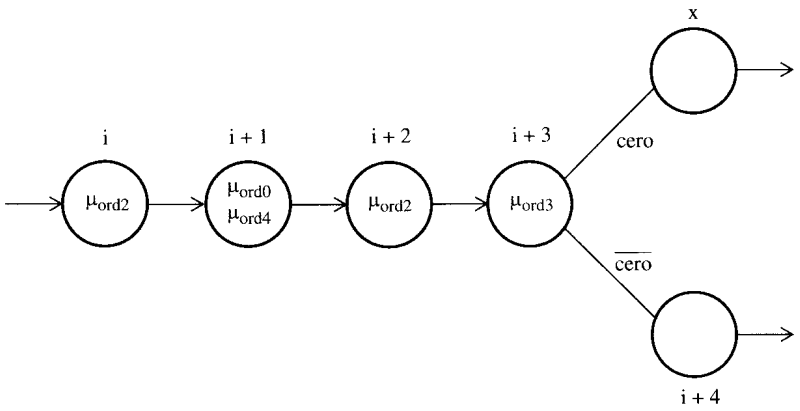


Figura 7.30: Grafo orientado del microprograma para la cuarta solución del ejemplo 7.3.1

Si se aprovecha la posibilidad dada por el procesador de ejecutar en paralelo varias microórdenes, con el empleo de la microprogramacion horizontal no codificada que proporcionaba la primera solución, es posible obtener un nuevo microprograma, con 4 microinstrucciones, que realiza la función pedida. En la Figura 7.30 se muestra el grafo orientado de este microprograma.

Las variantes expuestas en la primera solución (microprogramación horizontal no codificada) y en la tercera solución (microprogramación vertical) se aplican directamente a este nuevo microprograma. Sin embargo la microprogramación horizontal codificada de la segunda solución debe modificarse para permitir la ejecución en paralelo de las microoperaciones μ_{ord0} y μ_{ord4}.

Sección 7-4

Ejemplo de diseño microprogramado: multiplicador binario

Para ilustrar el diseño de una unidad de control microprogramada, se considera otra vez el multiplicador binario que ya se trató en el tema 5. Se necesitan conocer tres elementos básicos en el diseño de toda unidad de control microprogramada:

1) El número de bits que debe poseer el *registro de datos de la memoria de control* (*RMC*)
2) El número de bits que debe poseer el *registro de dirección de la memoria de control* (*RDC*)
3) El tamaño de la memoria de control

En primer lugar es preciso analizar las microoperaciones que se necesitan para realizar la multiplicación. En la Figura 7.31 se repite el diagrama ASM sin salida condicional deducido en el tema anterior (conviene recordar que este tipo de diagrama modela máquinas secuenciales tipo Moore que son las que se van a emplear en el diseño de la unidad de control microprogramada).

Dentro del conjunto de microoperaciones no se van a considerar (de la misma forma que se hizo ya en el diseño de la unidad de control con lógica cableada) la carga del multiplicando en el registro *M* y del multiplicador en el registro *m*. En la Tabla 5.2 del tema 5 se mostraba que sólo se necesitan cuatro señales de control en la unidad de procesamiento del multiplicador: *Inicializar*, *Sumar_Cargar*, *Desplazar_Dec* y *Borrar_C*. Estas señales de control y las transferencias de registros asociadas se muestran en la Tabla 7.7 .

Señal de control	Transferencia de registro	Estados en los que la señal está activa	Bit de posición en la μI	Nombre simbólico
Inicializar	$A \leftarrow 0, P \leftarrow n - 1$	S_{00}	0	IN
Sumar_Cargar	$A \leftarrow A + M, C \leftarrow c_a$	S_{10}	1	SC
Borrar_C	$C \leftarrow 0$	S_{00}, S_2	2	BC
Desplazar_Dec	$C \| A \| m \leftarrow C \| A \| m >> 1, P \leftarrow P - 1$	S_2	3	DD

Tabla 7.7: Señales de control y transferencias de registros asociadas para el multiplicador binario

Las cuatro señales de control se pueden utilizar tal como se dan o se pueden codificar para reducir el número de bits en la microinstrucción. Si no se codifican las señales de control se requieren cuatro bits (uno por cada señal de control) en el *campo de control* de la microinstrucción. Si por el contrario se codifican, se debe hacer con la suficiente flexibilidad para que se tomen en consideración todas las posibles combinaciones distintas de las señales de control que se necesitan. Si se desea utilizar una única palabra de

control para cada una de las combinaciones, hay que analizar cuáles son los diferentes conjuntos de señales de control que se emplean. De la Tabla 7.7 se deduce que en el multiplicador se dan las cuatro combinaciones siguientes:

1) *Inicializar* y *Borrar_C* en el estado S_{00}
2) *Sumar_Cargar* en el estado S_{10}
3) *Borrar_C* y *Desplazar_Dec* en el estado S_2
4) Ninguna señal de control activa en los estados S_0 y S_1

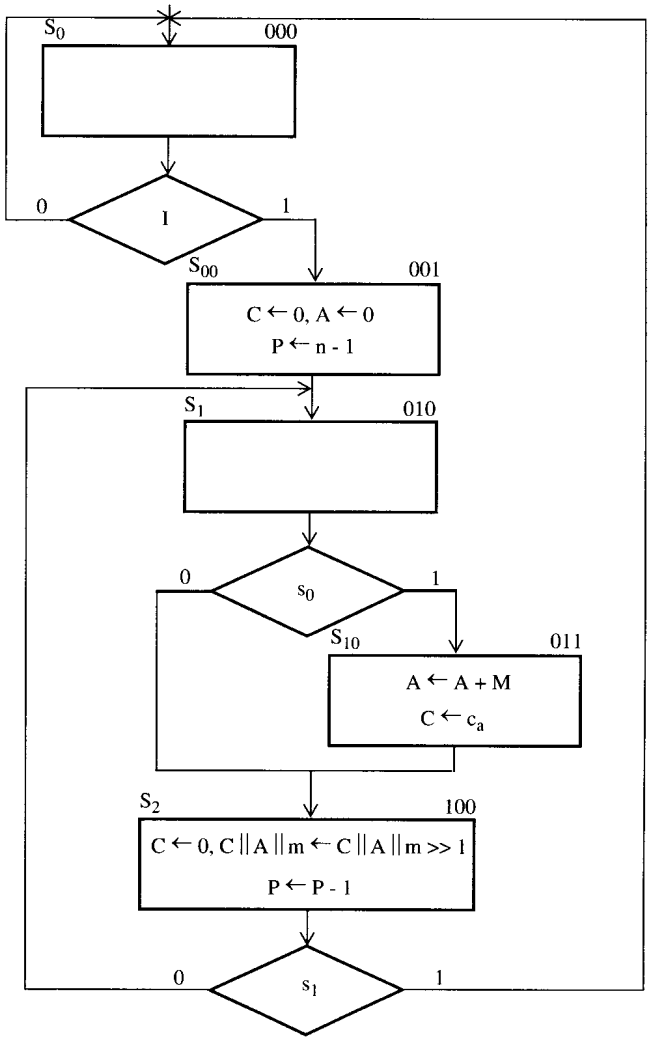


Figura 7.31: Diagrama ASM para el diseño de la unidad de control microprogramada del multiplicador

Como hay cuatro combinaciones distintas en la codificación se precisan dos bits en el campo de control de la microinstrucción para controlar la unidad de procesamiento del multiplicador. Si se hace esto, será necesario emplear un decodificador a la salida de la memoria de control para generar las señales de control originales. En ese caso, como sólo se ahorran dos bits y el número de estados es pequeño, la reducción en el tamaño de la ROM de la memoria de control no es suficiente para que se justifique la inclusión de un decodificador. Por este motivo no se codificarán las señales de control. En la cuarta columna de la Tabla 7.7 se muestran los bits de la microinstrucción que representan cada señal de control. En la quinta columna se da un nombre simbólico para cada una de las microoperaciones (se utilizarán posteriormente en la escritura de los microprogramas).

El resto de la microinstrucción se dedica al secuenciamiento de la unidad de control. Como ya se ha visto hay muchas alternativas para diseñar el secuenciador. El método de diseño escogido determina los campos que se necesitan en la microinstrucción. Como un primer paso en este diseño, se deben considerar los requisitos de secuenciamiento definidos por el diagrama ASM del multiplicador de la Figura 7.31. En los estados S_{00} y S_{10}, el próximo estado no depende de ninguna señal de condición ni de ninguna entrada. En el estado S_0, el próximo estado depende del valor de la señal de entrada *I*. En el estado S_1, el próximo estado depende de la señal de condición s_0. Finalmente en el estado S_2, el próximo estado depende de la señal de condición s_1. Para los casos en los que el próximo estado depende de una señal de entrada o una señal de condición, se necesitará una pareja de direcciones, una para cuando el valor de la entrada es 0 y otra para cuando es 1.

El método empleado para definir las direcciones es una decisión importante en el diseño del secuenciador. Existen dos formas típicas de hacerlo.

1) Incluir las dos direcciones en la microinstrucción que controla la decisión y según sea el valor de la variable de decisión, una de las dos direcciones se carga en *RDC*. Este método permite la asignación arbitraria de direcciones a los estados y asegura que no se necesita añadir ningún estado para proporcionar el secuenciamiento deseado. Sin embargo requiere que cada microinstrucción incluya dos direcciones lo que genera microinstrucciones muy anchas.

2) Utilizar un contador con capacidad de carga en paralelo como registro *RDC*. Una de las dos direcciones se obtiene de la microinstrucción y la otra simplemente incrementando el registro *RDC*. Este método requiere como máximo una dirección por microinstrucción. No obstante la asignación de las direcciones a los estados se complica y a veces es necesario añadir estados extras para proporcionar el secuenciamiento deseado. Estos estados pueden retardar la operación del sistema debido a que se necesitan ciclos de reloj adicionales para pasar por ellos.

La elección del método se basa en el compromiso que se establezca entre la velocidad y el coste del sistema. Para la unidad de control microprogramada del multiplicador se escoge el primer método de dos direcciones por microinstrucción que proporciona el diseño más simple.

Como se tienen cinco estados en la unidad de control del multiplicador (S_0, S_{00}, S_1, S_{10} y S_2) los dos campos de dirección de la microinstrucción (*Dirección*2 y *Dirección*1) necesitan tres bits cada uno de ellos. Además se requiere un campo de dos bits (*SEL*) para seleccionar si hay que realizar una decisión y en ese caso cual de las tres variables *I*, s_0 o s_1 va a controlar dicha decisión. Queda así definido el formato de la microinstrucción que se muestra en la Figura 7.32, que como se ve contiene 12 bits de los cuales se utilizan 4 para controlar la unidad de procesamiento y los otros 8 para el secuenciamiento de las microinstrucciones.

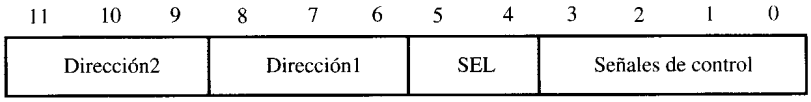


Figura 7.32: Formato de la microinstrucción

La Tabla 7.8 da la definición de los cuatro valores binarios del campo *SEL*. Si *SEL* es 00 (se representa de forma simbólica por *SIG*), la próxima dirección se obtiene del campo *Dirección1* de la microinstrucción. Si *SEL* es 01(se representa de forma simbólica por *DI*), se realiza una decisión que se basa en el valor de la señal *I*. Si *I* es 0 (1), la siguiente dirección que se carga en el registro *RDC* es la que está contenida en el campo *Dirección1* (*Dirección2*). Una conducta análoga se aplica a las decisiones sobre s_0 (se representa de forma simbólica por Ds_0) y sobre s_1 (se representa de forma simbólica por Ds_1). La información de la Tabla 7.8 completa la especificación de la unidad de control microprogramada.

SEL		Secuenciamiento de microórdenes
Nombre simbólico	**Código binario**	
SIG	00	RDC ← Dirección1
DI	01	$\overline{I}$ RDC ← Dirección1 I RDC ← Dirección2
Ds_0	10	$\overline{s}_0$ RDC ← Dirección1 s_0 RDC ← Dirección2
Ds_1	11	$\overline{s}_1$ RDC ← Dirección1 s_1 RDC ← Dirección2

Tabla 7.8: Definición del campo de selección SEL para el secuenciamiento de la unidad de control del multiplicador

Con la información de las Tablas 7.7 y 7.8, y de la Figura 7.32 se puede diseñar la unidad de control. La longitud de palabra de la memoria de control es de 12 bits que es la longitud de la microinstrucción. Como en la Figura 7.31 sólo hay cinco estados, la memoria de control contiene 5 palabras. Se requiere así un registro de dirección de la memoria de control (*RDC*) de 3 bits (con capacidad de carga en paralelo) para direccionar las 5 palabras.

Las direcciones que se cargan en *RDC* vienen de los campos *Dirección1* y *Dirección2* de la microinstrucción que en ese momento está en las líneas de salida de la memoria de control. Tres multiplexores de 2 a 1 (*MUX1*) con sus salidas conectadas a la entrada de datos del registro *RDC* permiten seleccionar entre estas dos fuentes. La señal de selección *c* para estos multiplexores se determina a partir de la Tabla 7.8. La selección de la variable de decisión o de la constante que se requiere para cada uno de los cuatro códigos *SEL* se realiza con un multiplexor de 4 a 1 (*MUX2*).

En la Figura 7.33 se muestra el diagrama de bloques que se obtiene para el multiplicador. La memoria de control es una ROM con una capacidad de cinco microinstrucciones de 12 bits. De los 12 bits de salida de la ROM, 4 se conectan a las entradas de control de la unidad de procesamiento, mientras que los restantes 8 bits determinan la dirección siguiente que se cargará en el registro *RDC*. Los bits del campo de selección (*SEL*)

controlan al multiplexor *MUX2*. Cuando *SEL* es igual a 00, *MUX2* selecciona la entrada 0 que tiene el valor 0. Un 0 en la línea de selección *c* del multiplexor *MUX1* selecciona *Dirección1* como la siguiente dirección tal como se especifica en la Tabla 7.8. Para *SEL* igual a 01, *MUX2* selecciona la entrada 1, que es *I*. Si *I* es 0, entonces la línea de selección *c* del multiplexor *MUX1* selecciona *Dirección1* como la próxima dirección, si *I* es 1 la dirección es *Dirección2*. Las otras variables de decisión s_0 y s_1 operan de una forma análoga.

En la Tabla 7.9 se da el microprograma del multiplicador binario en la notación de transferencia de registros. Esta descripción se corresponde con el diagrama ASM de la Figura 7.31. Se debe observar que en el microprograma hay una microinstrucción que se corresponde con cada uno de los estados del diagrama ASM. En este diagrama, el código binario de cada estado es el contenido del registro *RDC* para ese estado. El MUX1 de la Figura 7.31 está compuesto en realidad por 3 MUX de 2 a 1 de un sólo bit cada uno.

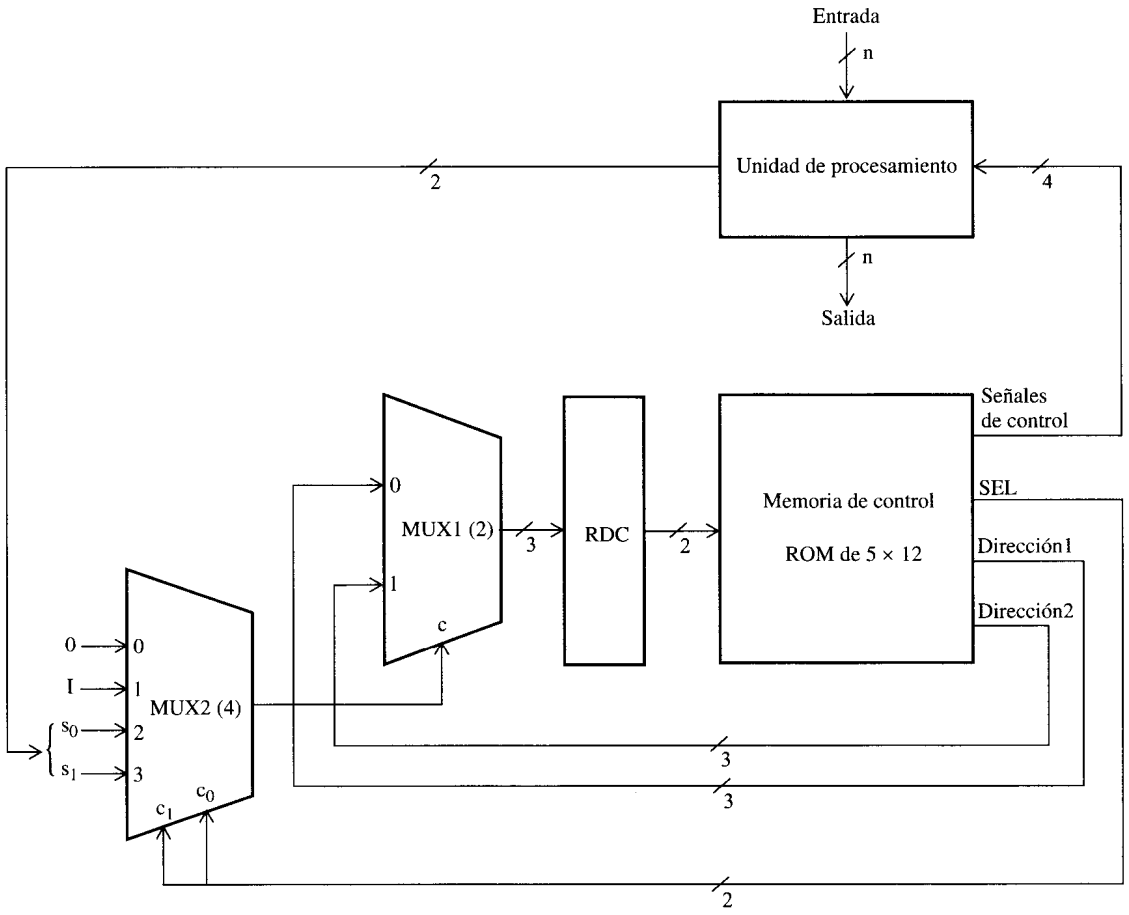


Figura 7.33: Unidad de control microprogramada del multiplicador

El microprograma en la notación de transferencia de registros se convierte en un microprograma simbólico sustituyendo cada transferencia de registro por los nombres dados a dichas operaciones y empleando identificadores (estados) para las direcciones (ver Tabla 7.10).

Dirección	Sentencias de transferencia simbólica
S_0	I: RDC $\leftarrow S_{00}$, $\overline{I}$: RDC $\leftarrow S_0$
S_{00}	C $\leftarrow$ 0, A $\leftarrow$ 0, P $\leftarrow$ n - 1, RDC $\leftarrow S_1$
S_1	s_0: RDC $\leftarrow S_{10}$, $\overline{s}_0$: RDC $\leftarrow S_2$
S_{10}	A $\leftarrow$ A + M, C $\leftarrow c_a$, RDC $\leftarrow S_2$
S_2	C $\leftarrow$ 0, C || A || m $\leftarrow$ C || A || m >> 1, s_1: RDC $\leftarrow S_0$, $\overline{s}_1$: RDC $\leftarrow S_1$, P $\leftarrow$ P - 1

Tabla 7.9: Microprograma del multiplicador en la notación de transferencia de registro

Dirección	Dirección2	Dirección1	Selección (SEL)	Señales de control
S_0	S_{00}	S_0	DI	Ninguna
S_{00}	---	S_1	SIG	IN, BC
S_1	S_{10}	S_2	Ds_0	Ninguna
S_{10}	---	S_2	SIG	SC
S_2	S_0	S_1	Ds_1	BC, DD

Tabla 7.10: Microprograma simbólico del multiplicador

En la Tabla 7.11, el microprograma simbólico se transforma en un microprograma binario al sustituir los nombres simbólicos por los correspondientes códigos binarios dados en la Figura 7.31 y en las Tablas 7.7 y 7.8. En el microprograma binario se han sustituido las entradas simbólicas no especificadas por una cadena de ceros.

Dirección	Dirección2	Dirección1	Selección (SEL)	Señales de control
000	001	000	01	0000
001	000	010	00	0101
010	011	100	10	0000
011	000	100	00	0010
100	000	010	11	1100

Tabla 7.11: Código binario del microprograma del multiplicador

Sección 7-5

Unidad de control microprogramada de un computador

En esta sección se describe un computador pedagógico sencillo SIMPLE2 que dispone de una unidad de control microprogramable *ucμP*. A este computador, que se basa en uno propuesto en el libro de A. Tanenbaum (1992), se le han realizado una serie de modificaciones para dotarle de una mayor funcionalidad. Está disponible un simulador de entrenamiento[1] de *ucμP*. El contenido que sigue es una adaptación del documento "*ucμP*: Simulador de un computador sencillo con unidad de control microprogramada. Versión 1.0".

El simulador permite ver cómo funciona el corazón de un computador: su unidad de control. Dado un conjunto de instrucciones máquina, el simulador carga en su memoria de control el conjunto de microinstrucciones que implementan dicho repertorio. El alumno tiene así una herramienta que permite *experimentar desde dentro* con el funcionamiento de un computador de forma dinámica. El simulador tiene otro objetivo secundario: una vez microprogramado un repertorio de instrucciones permite su funcionamiento a nivel de lenguaje máquina. De este modo, también se puede utilizar como un sencillo entrenador de lenguaje máquina sobre el que probar distintos algoritmos. Por ejemplo, puede ser útil para probar algoritmos de operaciones numéricas que no estén soportadas por la ALU (por ejemplo operaciones de multiplicación y división en coma flotante, etc.), con la ventaja adicional de que el repertorio de instrucciones de la máquina es redefinible a un nivel más bajo (nivel de microprogramación).

Ha sido diseñado con el objetivo de que lo utilicen los alumnos de la asignatura *Estructura y Tecnología de Computadores II* de la Escuela Universitaria de Informática de la UNED, aunque sin duda resultará también de interés para cualquiera que esté interesado en el estudio del funcionamiento de un computador digital. Se incluye en el CD-ROM que se distribuye gratuitamente a todos los alumnos que están matriculados en la citada asignatura.

7.5.1 Estructura de SIMPLE2

La estructura de SIMPLE2 puede dividirse en sus tres componentes fundamentales: unidad aritmético lógica y ruta de datos, unidad de control y memoria principal. En los apartados siguientes se describen cada una de estas unidades.

Unidad aritmético lógica y ruta de datos de SIMPLE2

La ruta de datos contiene las entrada y las salidas de la ALU, se encarga de la gestión de los cálculos, está controlada por la unidad de control y genera las señales de condición. En la Figura 7.34 se muestra la ruta de datos de SIMPLE2 que consta de los siguientes elementos:

1. El simulador de *ucμP* ha sido desarrollado por alumnos de la Escuela de Ingeniería Técnica en Informática de la UNED bajo la dirección del Prof. Dr. Ángel Pérez de Madrid.

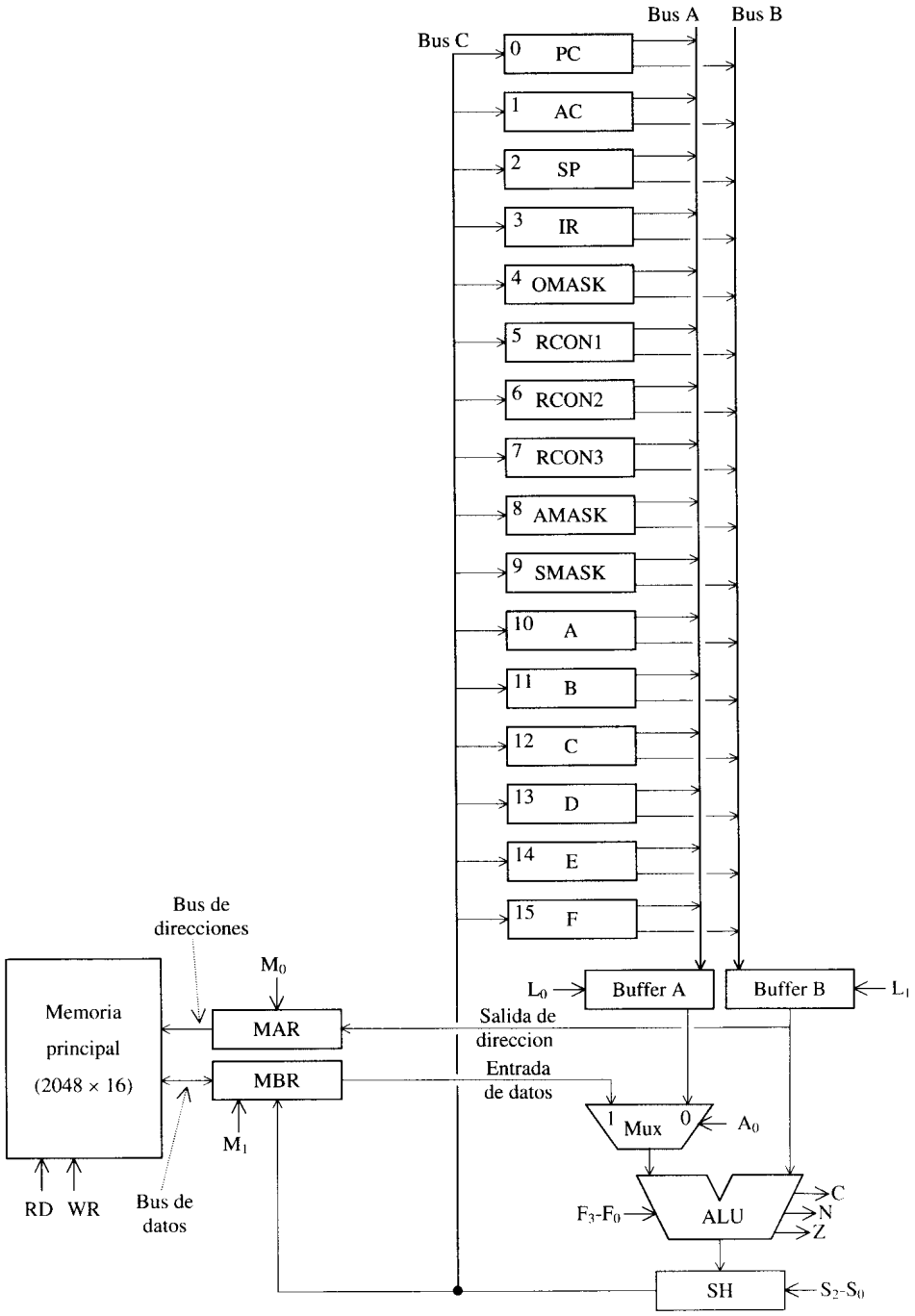


Figura 7.34: Ruta de datos de SIMPLE2 y conexión con la memoria principal

1) 16 *registros de* 16 *bits* (ver Tabla 7.12). Estos registros los tiene accesibles SIMPLE2 a nivel de microprogramación.

Registro	Función del registro
PC:	Contador de programa: contiene la dirección de memoria de la siguiente instrucción a ejecutar
AC:	Acumulador: sirve para cargar datos desde la memoria o hacia la memoria
SP:	Puntero de pila: dirección de la última posición de la pila en memoria
IR:	Registro de instrucción: contiene la instrucción máquina que se está ejecutando
OMASK:	Registro constante cargado con el valor $0000011111111111_2 = 2047_{10}$
RCON1:	Registro constante 1 cargado con el valor $0000000000000000_2 = 0_{10}$
RCON2:	Registro constante 2 cargado con el valor $0000000000000001_2 = +1_{10}$
RCON3:	Registro constante 3 cargado con el valor $1111111111111111_2 = -1_{10}$
AMASK:	Registro constante cargado con el valor $0000111111111111_2 = 4095_{10}$
SMASK:	Registro constante cargado con el valor $0000000011111111_2 = 255_{10}$
A, B, C, D, E, F:	Registros de propósito general

Tabla 7.12: Registros accesibles en SIMPLE2 a nivel de microprogramación

2) *Dos registros de* 16 *bits, Buffer A y Buffer B.* Sirven para retener la entrada a la ALU y se activan con las señales de control L_0 y L_1.

3) *Un multiplexor, Mux.* Controla la entrada izquierda de la ALU, con la señal de control A_0.

4) *Una unidad aritmético lógica (ALU).* Puede realizar seis operaciones (ver Tabla 7.13).

$F_3F_2F_1F_0$	Operación	Descripción de la salida
0 0 0 0	A + B	Suma aritmética de A y B (A procede de BufferA y B de BufferB)
0 0 0 1	$A \wedge B$	AND-lógico entre A y B
0 0 1 0	A	No hace nada, permite que A pase a través de la ALU (modo transparente)
0 0 1 1	$\overline{A}$	Complemento a 1 de A
0 1 0 0	$A \vee B$	OR-lógico entre A y B
0 1 0 1	$A \oplus B$	OR-exclusivo entre A y B

Tabla 7.13: Operaciones de la ALU de SIMPLE2

Las cuatro líneas de control de la ALU, F_3, F_2, F_1 y F_0 especifican la función a realizar, aunque sólo se utilizan las tres menos significativas, se han incluido cuatro para permitir desarrollos futuros. La ALU genera 3 bits de estado derivados de su salida: *N*, que se pone a 1 cuando la salida

de la ALU es negativa, *Z*, que se pone a 1 cuando la salida es 0 y *C*, que se pone a 1 cuando se produce un arrastre.

5) *Un registro de desplazamiento* (*SH*). Puede realizar tres funciones (ver Tabla 7.14). Aunque sólo se utilizan dos líneas de control, se han incluido tres para permitir ampliaciones futuras.

S_2 S_1 S_0	Descripción de la salida
0 0 0	No desplaza.
0 0 1	Desplaza un bit a la derecha.
0 1 0	Desplaza un bit a la izquierda.

Tabla 7.14: Operaciones del registro de desplazamiento

6) *Un registro de datos de memoria* (*MBR*). Este registro, junto con el *MAR*, sirve para la comunicación con la memoria. Su función es almacenar los datos que haya que cargar en memoria o bien almacenar los datos procedentes de memoria. La señal de control M_1 controla la carga de *MBR* a partir de la salida del registro de desplazamiento.

7) *Un registro de dirección de memoria* (*MAR*). Contiene la dirección de memoria donde hay que leer o escribir el dato que contiene el *MBR*. La señal de control M_0 controla la carga del registro.

8) *Tres buses* (*BusA, BusB y BusC*). Los buses *A* y *B* alimentan la ALU y el bus *C* carga la salida de *SH* a los registros y/o al *MBR*.

Cada registro puede volcar su contenido a uno de los buses *A* o *B* (o ambos a la vez) y tomarlo del bus *C*. Los registros *PC, AC, SP, IR, OMASK, RCON1, RCON2, RCON3, AMASK, SMASK, BufferA, BufferB, MAR* y *MBR* son registros de propósito especial mientras que los registros *A,B,C,D,E* y *F* son de propósito general.

Unidad de control microprogramable *ucμP* de SIMPLE2

La unidad de control microprogramable *ucμP* de SIMPLE2 consta de los siguientes componentes:

- *Reloj*. Marca la cronología de los sucesos dentro de *ucμP*, y en este caso se divide en cuatro subciclos. Posteriormente se verá con más detalle el generador de subciclos de reloj.
- *Memoria de control*. Formada por 1024 palabras de 48 bits. La memoria de control necesita dos registros; el registro de direcciones de memoria de control (*RDC*) de 11 bits que indica la dirección de la próxima microinstrucción (en la memoria de control), y el registro de datos de memoria de control (*RMC*) de 48 bits que contiene la microinstrucción en curso. Es importante darse cuenta de que la memoria de control y la memoria principal son totalmente diferentes. La primera almacena los microprogramas, que constituyen el repertorio de instrucciones máquina, y la segunda, el programa a nivel máquina convencional.
- *Secuenciador*. Determina la siguiente microinstrucción que hay que extraer, dependiendo de las señales de control (*FIR*) y de condición (*N, Z, C*) (ver Figura 7.35).

Según se puede comprobar en la Figura 7.35, una instrucción máquina consta de un campo código de operación (5 bits) y de un campo operando (11 bits). El código de operación, tal como está construido el secuenciador, se puede consultar para determinar la dirección de la próxima microinstrucción.

En la Tabla 7.15 se muestra como el multiplexor del secuenciador *MMUX* selecciona la próxima microinstrucción. Así, siempre que el campo *FIR* de *RMC* sea 1 se seleccionará la dirección indicada en *IR*[*cod_op*], es decir, se saltará al microprograma correspondiente a la próxima instrucción, independientemente de lo que indique el bloque de *lógica de bifurcación*.

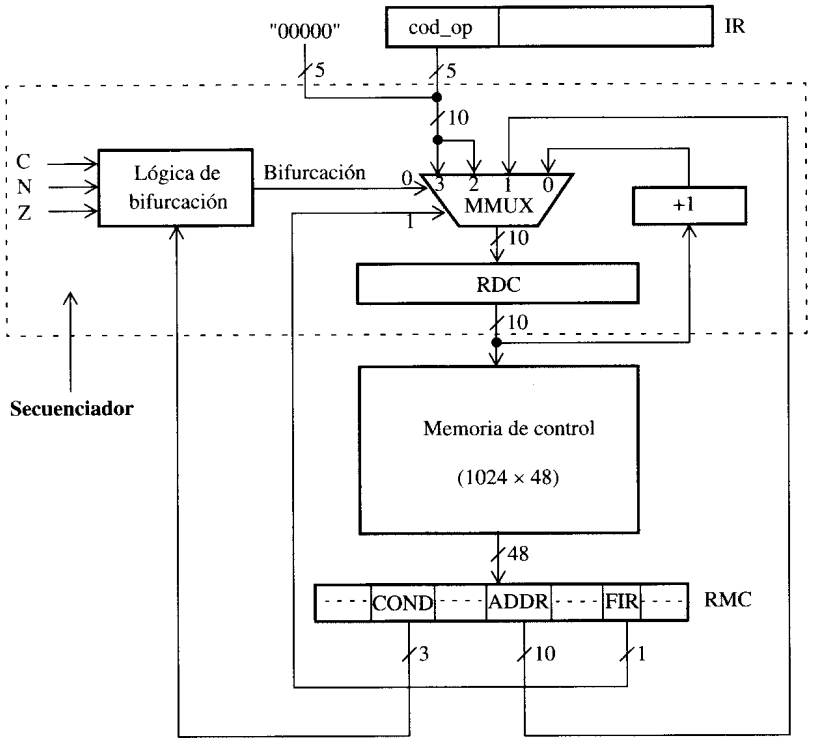


Figura 7.35: Secuenciador de microprograma (se incluye la memoria de control, *IR* y *RMC*).

Entradas de control a MMUX		Próxima dirección
1 (FIR)	0 (Bifurcación)	
0	0	*RDC + 1*
0	1	*RMC[ADDR]*
1	0	*IR[cod_op]*
1	1	*IR[cod_op]*

Tabla 7.15: Selección de la próxima microinstrucción mediante el multiplexor MMUX del secuenciador

El campo *COND* de *RMC* viene dado por los tres bits $C_2\ C_1\ C_0$. La función lógica que implementa el bloque de lógica de bifurcación deducida a partir de este campo es:

$$\text{Bifurcación} = \overline{C}_2\overline{C}_1C_0N + \overline{C}_2C_1\overline{C}_0Z + \overline{C}_2C_1C_0C + C_2\overline{C}_1\overline{C}_0$$

En la Tabla 7.16 se muestra el significado individual de los bits del campo *COND*.

$C_2\,C_1\,C_0$	Acción
0 0 0	No salta.
0 0 1	Salta si N = 1.
0 1 0	Salta si Z = 1.
0 1 1	Salta si C = 1
1 0 0	Salta siempre

Tabla 7.16: Tipos de bifurcaciones de *ucμP*

Memoria principal de SIMPLE2

La memoria principal contiene 2048 palabras de 16 bits. Es accesible a través de los registros *MBR* y *MAR*, siendo el primero donde se almacena el dato y el segundo el que indica la dirección de memoria de dicho dato.

Un ciclo de lectura o escritura dura dos ciclos, es decir, si se quiere leer o escribir un dato en la memoria se necesitará tener activado el campo *MAR* o *WR* durante dos microinstrucciones seguidas. Al final de la segunda microinstrucción el dato se habrá leído o escrito. Durante este tiempo se pueden realizar otras microoperaciones, siempre que éstas no interfieran con la operación de memoria.

7.5.2 Formato de la microinstrucción de *ucμP*

Cada microinstrucción de *ucμP* tiene 48 bits en formato vertical (ver Tabla 7.17). La Tabla 7.18 muestra cómo se definen los campos de la microinstrucción de *ucμP*.

AMUX	COND			SH			MBR	MAR	RD	WR	ENC	C				B				A			
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24

ADDR										FIR	ALU				NO UTILIZADOS								
23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Tabla 7.17: Formato de las microinstrucciones

Campo de la μI		Acción	
AMUX:	Controla la entrada izquierda de la ALU	0: BufferA	1: MBR
COND:	Control del salto	Ver Tabla 7.16	
SH:	Función del registro de desplazamiento	Ver Tabla 7.14	
MBR:	Carga de MBR desde el registro de desplazamiento	0: No carga	1: Carga
MAR:	Carga de MAR desde BufferB	0: No carga	1: Carga
RD:	Petición de lectura de memoria	0: No pide	1: MBR ← M[MAR]
WR:	Petición de escritura en memoria	0: No pide	1: M[MAR] ← MBR
ENC:	Habilita el bus C (control de almacenamiento)	0: No almacena	1: Almacena
C:	Selecciona el registro donde almacenar el dato procedente de la ALU. ENC debe estar activada a 1.	0000: PC 0010: SP	0001: AC .... etc
B:	Selecciona la fuente del bus B	0000: PC 0010: SP	0001: AC .... etc
A:	Selecciona la fuente del bus A	0000: PC 0010: SP	0001: AC .... etc
ADDR:	Dirección de la siguiente microinstrucción		
FIR:	Controla la carga del código de operación (5 bits) de la instrucción máquina almacenada en IR en RDC	0: No se carga	1: Se carga
ALU:	Función de la ALU	Ver Tabla 7.13	

Tabla 7.18: Definición de los campos de la microinstrucción de *ucμP*

Como se puede observar no se han utilizado los 9 bits menos significativos de la microinstrucción que quedan reservados para ampliaciones futuras de *ucμP*.

7.5.3 Fases en la ejecución de una microinstrucción

Uno de los aspectos más importantes es la cronología de las microinstrucciones. Para que *ucμP* funcione correctamente, es necesario seguir un orden en la línea de ejecución de las microinstrucciones. Por ejemplo, un ciclo de la ALU consiste en cargar los registros *BufferA* y *BufferB*, darle tiempo a la ALU para operar y almacenar el resultado en el registro correspondiente. Pero, ¿que ocurriría si en lugar de cargar previamente los registros en los buses, se hubiese primero operado con la ALU? Cabría esperar que el contenido del registro de resultado contuviese un valor que nada tiene que ver con el valor esperado, ya que se habría operado con datos erróneos.

Evidentemente hay que seguir una secuencia a la hora de ejecutar las microinstrucciones y para ello, se consideran cuatro subciclos de reloj. En cada subciclo de reloj se ejecutarán una serie de microoperaciones. Se debe recordar que una microinstrucción puede contener varias microoperaciones. Los subciclos de reloj en *ucμP* son los siguientes:

- *Subciclo* 1. Carga la siguiente microinstrucción a ejecutarse en el registro *RMC*.
- *Subciclo* 2. Saca el contenido de los registros a los buses A y B, y los almacena en los registros *BufferA* y *BufferB*.
- *Subciclo* 3. Espera a que la ALU y el registro de desplazamiento (*SH*) operen y produzcan una salida estable y si es necesario carga el registro de direcciones de memoria (*MAR*).
- *Subciclo* 4. Almacena el resultado del registro desplazamiento (*SH*), en el banco de registros y si es necesario en el registro de datos de memoria (*MBR*).

La Figura 7.36 es un diagrama de bloques detallado de la microarquitectura de *ucμP* donde se han incluido los subciclos de reloj.

7.5.4 Inicialización de SIMPLE2

Al ponerse en funcionamiento SIMPLE2, *PC* y *RDC* se ponen a 0 automáticamente. Por este motivo, la primera microinstrucción a ejecutar será la que tiene dirección 0. En esta dirección de la memoria de control se deberá encontrar el microprograma de búsqueda de la primera instrucción máquina (que en principio, si no se modifica *PC*, será la que tiene dirección 0), o un salto incondicional a este microprograma.

Además, el puntero de la pila *SP*, toma el valor 0 y las tres señales de condición, *N, Z* y *C* se ponen también a 0. Los registros constantes (*OMASK*, *RCON1*, *RCON2*, *RCON3*, *AMASK* y *SMASK*) toman sus valores correspondientes. Todas estas acciones podrían requerir un microprograma de arranque específico. Sin embargo, por sencillez, no se tendrá en cuenta esta posibilidad.

7.5.5 Búsqueda y decodificación de las instrucciones máquina

La unidad de control de un computador es una máquina secuencial y de forma cíclica realiza una serie de operaciones:

- Búsqueda de la siguiente instrucción máquina.
- Decodificación.
- Búsqueda de los operandos.
- Ejecución.
- Almacenamiento de resultados.
- Determinación de la dirección de la siguiente instrucción máquina.

Estas operaciones se agrupan en las fases que forman el *ciclo de instrucción* (no se considera la gestión de interrupciones). Algunas de estas fases, como son la búsqueda y decodificación, son comunes a todas las instrucciones máquina, por lo que se puede utilizar un único microprograma para cada una de ellas. Otras, como es la ejecución, depende de cada instrucción. Y en algunas de estas fases existen microsubrutinas que pueden reutilizarse en varias partes del código de control. En este apartado se va a considerar cómo se pueden resolver las dos primeras fases en SIMPLE2.

Búsqueda de la siguiente instrucción máquina

Esta tarea es sencilla. Habrá que hacer un acceso a la dirección de memoria principal indicada por el contador de programa (*PC*), y almacenar su contenido en el registro de instrucción (*IR*).

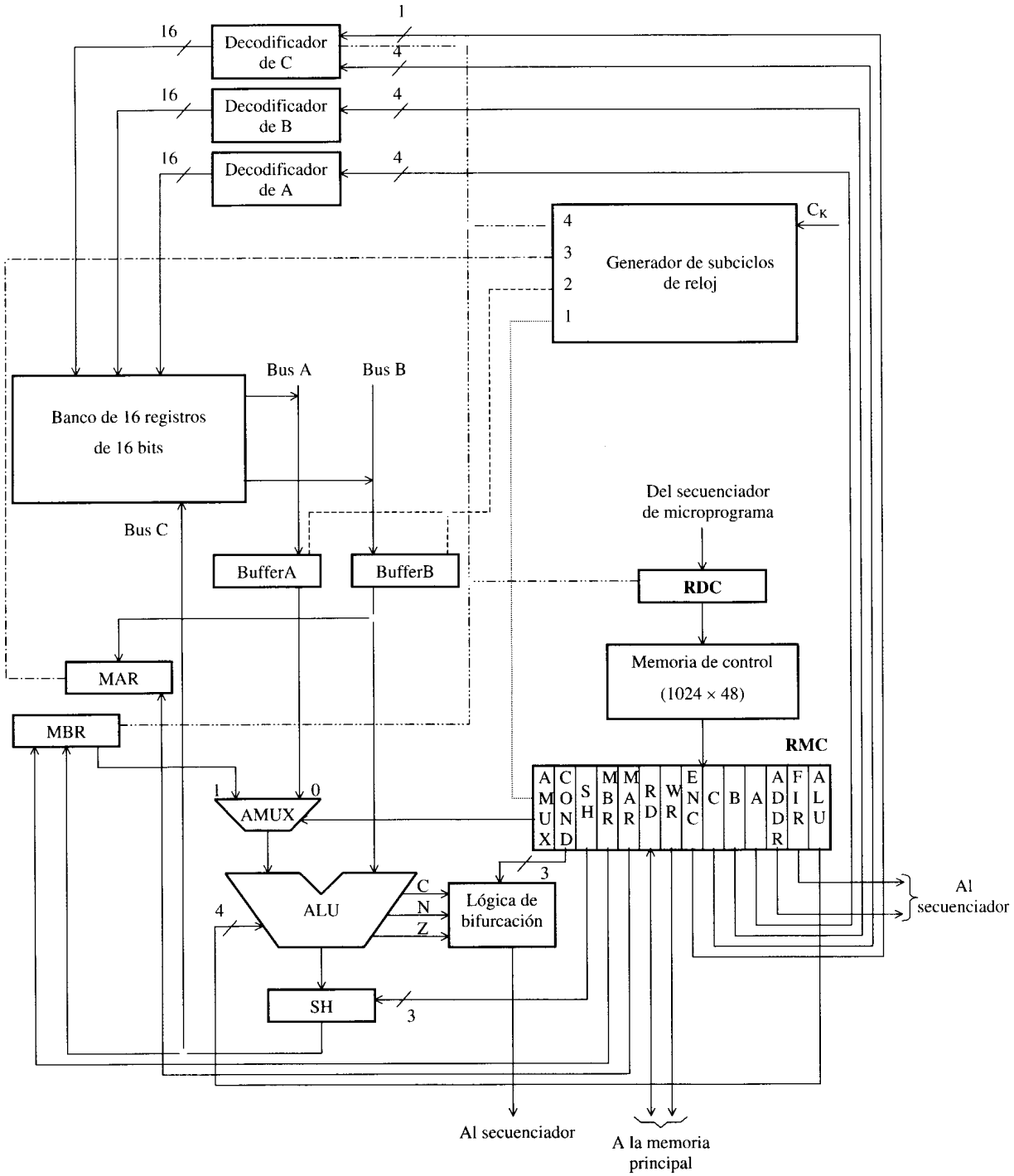


Figura 7.36: Microarquitectura de SIMPLE2 que muestra el generador de subciclos de reloj

Decodificación

Consiste en obtener a partir del código de operación la dirección dentro de la memoria de control en la que se encuentra el microprograma correspondiente a cada instrucción máquina. En la Figura 7.35 se observa que el campo código de operación del registro de instrucción *IR* puede cargarse en el registro de dirección de la memoria de control *RDC*. De esta manera, el código de operación determina una posible dirección de inicio para su microprograma. Ahora bien, si la instrucción cuyo código de operación es n determina la dirección n, la instrucción cuyo código es $n + 1$ determinará la dirección $n + 1$. Se tiene por tanto únicamente una dirección en la memoria de control asociada a cada instrucción máquina, lo que no deja margen para contener los respectivos microprogramas. Por eso, *en la dirección n se almacenará un salto incondicional a la dirección en la que realmente comienza el microprograma* (ver Figura 7.37).

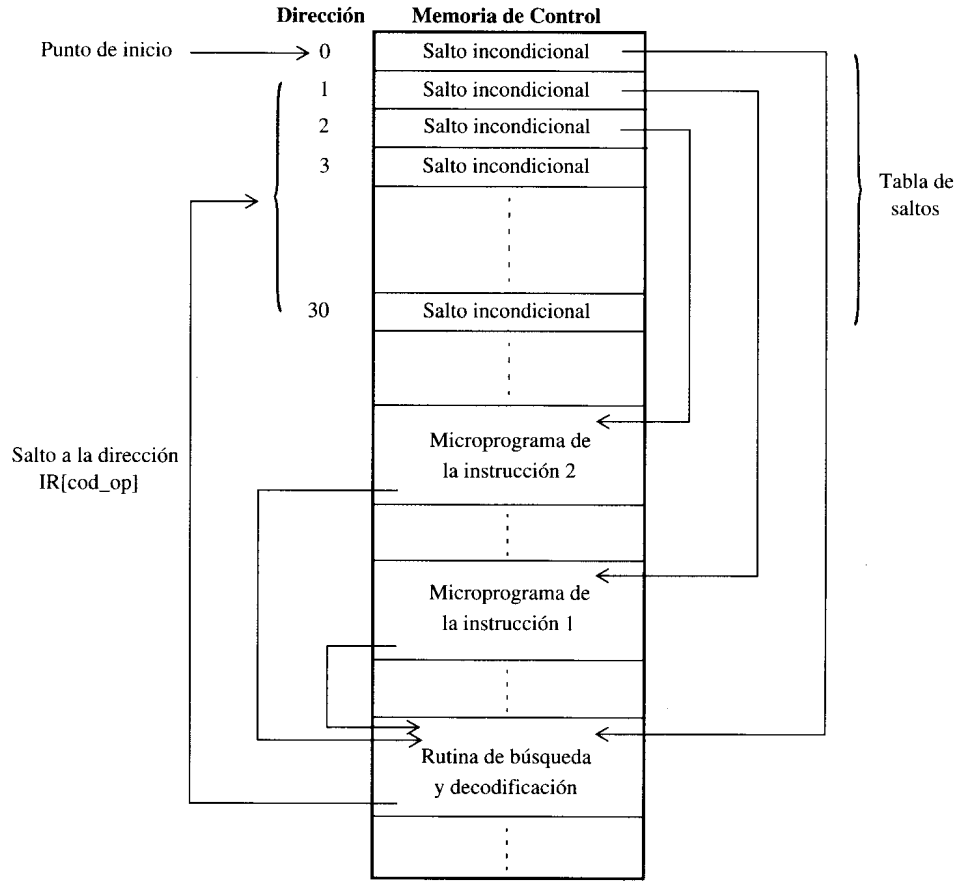


Figura 7.37: Organización de la memoria de control

Conviene tener presente además, que al arrancar el computador, *RDC* se inicializa automáticamente a 0. Si se adopta el criterio de que ninguna instrucción máquina puede tener un código de operación igual a 0, la dirección 0 de la memoria de control puede contener un salto incondicional al microprograma de búsqueda y decodificación de instrucciones máquina.

La instrucción de código de operación 31_{10} también está reservada y no puede ser utilizada. Para indicar el final de cualquier programa existirá en todos los repertorios de instrucciones una instrucción especial de parada. Esta instrucción es *HALT* y su código de operación es $11111_2 = 31_{10}$ (el campo operando es irrelevante). Esta instrucción se trata de manera especial por la unidad de control del procesador y no tiene que ser microprogramada. Por este motivo el máximo número de instrucciones que puede tener la máquina es 30 ($2^5 = 32$, menos 2 que han sido reservadas). De esta manera, la memoria de control queda organizada como se muestra en la Figura 7.37. Obsérvese que al final del microprograma correspondiente a cada instrucción máquina se salta al microprograma de búsqueda de la siguiente instrucción.

7.5.6 Ejemplo: Microprogramación de un repertorio de instrucciones máquina de SIMPLE2

En la Tabla 7.19 se muestra un posible repertorio de instrucciones máquina que es capaz de interpretar la unidad de control *ucμP* de SIMPLE2. En este apartado y a modo de ejemplo se realiza la microprogramación de algunas instrucciones. De forma análoga se pueden hacer el resto de las instrucciones del repertorio.

Binario	Nemotécnico	Instrucción	Significado
00001xxxxxxxxxx	LODD	Carga directa	AC ← M[x]
00010xxxxxxxxxx	LODI	Carga inmediata	AC ← x
00011xxxxxxxxxx	STOD	Almacenamiento directo	M[x] ← AC
00100xxxxxxxxxx	ADDD	Suma directa	AC ← AC + M[x]
00101xxxxxxxxxx	ADDI	Suma inmediata	AC ← AC + x
00110xxxxxxxxxx	SUBD	Resta directa	AC ← AC - M[x]
00111xxxxxxxxxx	SUBI	Resta inmediata	AC ← AC - x
01000	PUSH	Apilar	SP ← SP - 1 M[SP] ← AC
01001	POP	Desapilar	AC ← M[SP] SP ← SP + 1
01010xxxxxxxxxx	JNEG	Salta si negativo	if AC<0 then PC ← x
01011xxxxxxxxxx	JZER	Salta si cero	if AC=0 then PC ← x
01100xxxxxxxxxx	JCAR	Salta si acarreo	if C=1 then PC ← x
01101xxxxxxxxxx	JUMP	Salto incondicional	PC ← x
01110xxxxxxxxxx	CALL	Subrutina	SP ← SP – 1 M[SP] ← PC PC ← x
01111	RETN	Vuelta subrutina	PC ← M[SP] SP ← SP + 1
11111	HALT	Parada del programa	

Tabla 7.19: Repertorio de instrucciones máquina y su significado

Microprograma de las fases de búsqueda y decodificación

La dirección de la siguiente instrucción máquina, salvo en las instrucciones de salto (condicional o incondicional) que habrá que tratar aparte, se obtiene incrementando el contador de programa: $PC \leftarrow PC + 1$. Se puede definir un pequeño microprograma que se encargue de esta operación (en adelante, por brevedad, sólo se indicarán los campos del formato de la microinstrucción que son relevantes en cada caso; todos los que no se mencionen explícitamente se suponen a cero).

Una vez incrementado *PC*, lo que queda es traer el contenido de la dirección a la que apunta desde la memoria principal al registro de instrucción *IR* y saltar incondicionalmente a la dirección de la memoria de control indicada por *IR*[*cod_op*]. Aquí se encontrará la dirección del microprograma correspondiente a la instrucción y queda así completada las fases de búsqueda y decodificación. En la Tabla 7.20 se muestra el microprograma de las fases de búsqueda y decodificación en notación simbólica y en hexadecimal.

Microinstrucción	Señales	Microoperaciones	μI en hexadecimal
PC ← PC + 1	A = 0000 B = 0110 AMUX = 0 ALU = 0000 SH = 000 ENC = 1 C = 0000 COND = 000	Selecciona PC como fuente del bus A Selecciona RCON2 (contenido = +1) como fuente del bus B Selecciona BufferA Operación de suma No desplaza Habilita el bus C para carga de registro Selecciona almacenar en PC No bifurca	0010 6000 0000
MAR ← PC; RD = 1	B = 0000 MAR = 1 RD = 1 COND = 000	Selecciona PC como fuente del bus Carga MAR con la dirección indicada por PC Inicio de la operación de lectura No bifurca	00C0 0000 0000
RD = 1; MBR ← M[MAR]; IR ← MBR	RD = 1 AMUX = 1 ALU = 0010 SH = 000 ENC = 1 C = 0011 COND = 000	Fin de la operación de lectura, MBR es válido Selecciona MBR como entrada a la UAL Modo transparente No desplaza Habilita el bus C para carga de registros Se carga el registro IR con la salida de SH No bifurca	8053 0000 0400
MPC ← IR[cod_op]	FIR = 1	Salto a la dirección de la tabla de saltos correspondiente	0000 0000 2000

Tabla 7.20: Microprograma en notación simbólica de las fases de búsqueda y decodificación

La rutina de búsqueda y decodificación consta, por tanto, de cuatro microinstrucciones que ocupan cuatro direcciones consecutivas en la memoria de control. Para un mejor aprovechamiento de la memoria de control se reservan sus cuatro últimas direcciones (1020 - 1023) para esta rutina.

Instrucción LODI n

Esta instrucción carga (*LOD*) el acumulador en modo inmediato (*I*) con el valor *n* (contenido en el campo de operando del registro de instrucción): $AC \leftarrow IR[operando]$.

La principal dificultad a la hora de microprogramar esta instrucción consiste en extraer *n* de *IR*. El formato de cada instrucción es *cccc cooo oooo oooo*, donde *c* representa un bit (0 ó 1) del código de

operación y cada *o* representa un bit del operando. Una vez que se ha extraído el operando, éste debe ser 0000 0*ooo oooo oooo*. Este resultado se puede obtener realizando el AND-lógico entre *IR* y la máscara 0000 0111 1111 1111 que es precisamente el valor del registro *OMASK*. Por tanto, $n = IR \wedge OMASK$, resultado que se puede guardar en cualquier registro. En este caso, se debe guardar el resultado en *AC*. En la Tabla 7.21 se muestran las microoperaciones que hay que realizar para ejecutar la instrucción *LODI n*.

Microinstrucción	Señales	Microoperaciones	µI en hexadecimal
AC ← IR[operando]	A = 0011	Selecciona IR como fuente del bus A	4011 43FF 0200
	B = 0100	Selecciona OMASK como fuente del bus B	
	AMUX = 0	Selecciona BufferA como entrada a la ALU	
	ALU = 0001	Operación AND	
	SH = 000	No desplaza	
	ENC = 1	Habilita el bus C para carga de registros	
	C = 0001	Se carga el registro AC con la salida de SH	
	COND = 100	Salto incondicional	
	ADDR = 1020_{10}	A la dirección 1020_{10}	

Tabla 7.21: Fase de ejecución de la instrucción LODI n

Los campos que no se utilizan no se han indicado y se supone que están a 0. Un valor diferente podría ocasionar problemas en algunos casos (por ejemplo, modificar inadvertidamente el valor de un registro). Al finalizar la microinstrucción se incluye un salto incondicional a la rutina de búsqueda y decodificación de la siguiente instrucción máquina, que según lo dicho anteriormente se supone que está en la dirección 1020_{10} de la memoria de control.

La extracción del operando a partir de *IR* es una tarea que se repite con frecuencia. Cabe pensar que podría ser interesante dedicar una microsubrutina a este fin. Sin embargo, puesto que puede realizarse en una única microinstrucción no merece la pena hacer esto y, por tanto, cada vez que se necesite extraer el operando se repetirá la operación que se ha mostrado aquí.

Instrucción PUSH

Como se indicó en el apartado 7.5.4, al inicializar SIMPLE2, el puntero de pila *SP* se pone a 0. Según la Tabla 7.19, la instrucción de apilar *PUSH* decrementa *SP* antes de colocar el dato en la pila. Por este motivo al apilar el primer dato resulta *SP* = -1, que se interpreta por la unidad de control como *SP* = 2047, que es la última dirección de memoria (téngase en cuenta que las direcciones de memoria no tienen signo). De esta forma, la pila comienza en la última dirección de la memoria, 2047, y crece hacia las direcciones más bajas de la misma.

Cuando se saca un dato de la pila se incrementa *SP*. El último dato se encuentra en la dirección 2047. Al sacarlo e incrementar *SP* resulta *SP* = 0 (recuérdese que 2047 es la última dirección de la memoria principal). Por este motivo, *SP* = 0 es el indicador de que la pila se encuentra vacía (en efecto, al arrancar SIMPLE2 la pila se encuentra vacía y *SP* = 0). Desde el punto de vista de la pila, la memoria principal se puede considerar un anillo en el que la primera dirección (0) y la última (2047) son consecutivas.

Como la pila crece hacia las direcciones bajas de la memoria principal, es posible que invada un área de memoria reservada a datos o instrucciones, con resultados catastróficos. Es responsabilidad del programador evitar esta situación. Algunas razones para el crecimiento excesivo de la pila incluyen entre otras errores en la microprogramación o en el programa en lenguaje máquina, pero también se puede producir con un

programa máquina correcto, aunque descuidado, que no ha tenido en cuenta el tamaño máximo que puede alcanzar la pila.

La instrucción *PUSH* en primer lugar decrementa el valor de *SP* (*SP* ← *SP* - 1) y a continuación carga en la posición de memoria a la que apunta el puntero de pila el contenido del acumulador (*M*[*SP*] ← *AC*). En la Tabla 7.22 se muestran las microoperaciones que hay que realizar para ejecutar la instrucción *PUSH*.

Microinstrucción	Señales	Microoperaciones	μI en hexadecimal
SP ← SP - 1	A = 0010 B = 0111 AMUX = 0 ALU = 0000 SH = 000 ENC = 1 C = 0010 COND = 000	Selecciona registro SP como fuente del bus A Selecciona registro RCON3 como fuente del bus B Selecciona BufferA como entrada a la ALU Operación suma No desplaza Habilita bus C para carga de registros Se carga el registro SP con la salida de SH No bifurca	0012 7200 0000
MAR ← SP	B = 0010 MAR = 1	Selecciona SP como fuente del bus B Carga registro MAR con la dirección indicada en SP	0080 2000 0000
MBR ← AC	A = 0001 AMUX = 0 ALU = 0010 SH = 000 MBR = 1	Selecciona registro AC como fuente del bus A Selecciona bufferA como entrada a la ALU Modo transparente No desplaza Carga registro MBR con la salida de SH	0100 0100 0400
MP[SP] ← AC	WR = 1	Primer ciclo de escritura (iniciar)	0020 0000 0000
MP[SP] ← AC	WR = 1 COND = 100 ADDR = 1020_{10}	Segundo ciclo de escritura (finalizar) Salto incondicional A la dirección 1020_{10}	4020 00FF 0000

Tabla 7.22: Fase de ejecución de la instrucción PUSH

Sección 7-6

Conclusiones

Un computador es una máquina programable de propósito general. El programador ve al computador como una máquina que comprende y ejecuta instrucciones de un lenguaje de alto nivel. Sin embargo, en último término, sólo es un conjunto de circuitos electrónicos. Entre estas dos visiones extremas hay un sistema de elevada complejidad. Para entender mejor la estructura de un computador, se puede suponer que existe una jerarquía de *niveles de abstracción*: (ver Figura 7.38)

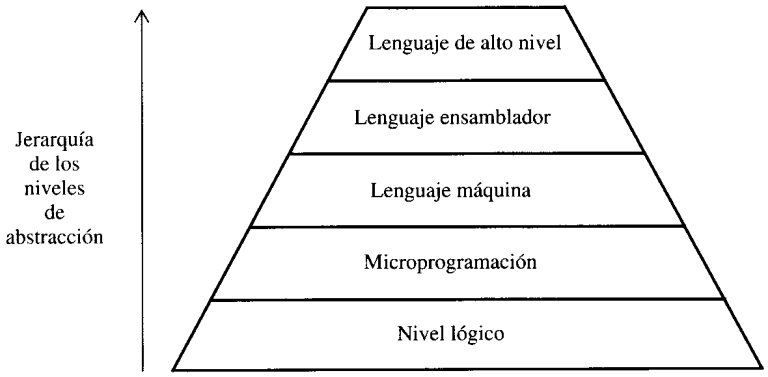


Figura 7.38: Niveles de interpretación de un computador

1) Nivel lógico
2) Nivel de microprogramación
3) Nivel de lenguaje máquina
4) Nivel de lenguaje ensamblador
5) Nivel de lenguaje de alto nivel

Un *intérprete* es un sistema que analiza y ejecuta instrucciones del lenguaje que describe un determinado nivel. Cada nivel de abstracción de un computador es idealmente ejecutado por un interprete realizado en el nivel inmediatamente superior de la jerarquía. Así el nivel de microprogramación es interpretado por un interprete realizado en el nivel lógico (es decir con circuitos digitales). Análogamente el nivel de lenguaje máquina lo interpreta el nivel de microprogramación y así sucesivamente hasta llegar a los lenguajes de alto nivel.

La unidad de control de un computador se encarga de extraer de la memoria cada instrucción, decodificarla y generar la secuencia correcta de señales de control para que pueda ejecutarse. En otras palabras la *unidad de control interpreta el lenguaje máquina de un computador* y por lo tanto, está constituida por los niveles lógicos y de microprogramación. Un *intérprete del nivel de microprogramación* interpreta las instrucciones de un lenguaje de microprogramación, llamadas microinstrucciones, directamente sobre los circuitos digitales del nivel lógico.

Un *intérprete del nivel de lenguaje máquina* interpreta las instrucciones del lenguaje máquina del computador con la ayuda de una unidad de control microprogramada. Este esquema no siempre se respeta, sino que en algunos computadores se suprime el nivel de microprogramación, interpretándose el lenguje máquina directamente sobre el nivel lógico. En este caso, se habla de una *unidad de control con lógica cableada*, mientras que en el anterior se dispone de una *unidad de control con lógica microprogramada*. Históricamente estas dos concepciones son las que se han utilizado en el diseño de la unidad de control de un computador.

1) *Unidad de control con lógica cableada*

   Se concibe la unidad de control como un circuito secuencial que genera ciertas secuencias fijas de señales de control. Los objetivos de diseño son reducir la circuitería que se necesita y aumentar la velocidad de trabajo del sistema.

2) *Unidad de control con lógica microprogramada*

   Los dos elementos centrales de una unidad de control microprogramada son la memoria de control y el secuenciador que proporciona la lógica necesaria para seleccionar la dirección de la siguiente microinstrucción.

   Alrededor de 1950 M. V. Wilkes planteó la necesidad de lograr un método flexible y sistemático para diseñar la unidad de control de un computador. El nuevo enfoque que denominó microprogramación trata una sentencia de la forma:

   $$activar\ las\ señales\ de\ control\{c_1, c_2, ..., c_k\}$$

   como una instrucción almacenada en una memoria especial denominada memoria de control. Estas instrucciones que se guardan en la memoria de control son las microinstrucciones y el conjunto de todas ellas, que realiza una instrucción máquina del computador, es el microprograma asociado a dicha instrucción. Un *microprograma* es pues una secuencia de microinstrucciones que interpreta una instrucción del lenguaje máquina de un computador. La secuencia de microinstrucciones de los diferentes ciclos de una instrucción finalizan con una microoperación de bifurcación, que indica en que posición de la memoria de control se continúa la ejecución del microprograma correspondiente.

   En una unidad de control microprogramada las señales de control se generan por programa y resulta sencillo introducir modificaciones en el diseño, ya que sólo se necesita alterar el contenido de la memoria de control. Cuando la memoria de control es de tipo RAM se dice que la unidad de control es microprogramable.

Uno de los aspectos más interesantes de la microprogramación es el hecho de que un mismo procesador (si es microprogramable) puede ejecutar varios microprogramas distintos y así proporcionar varios lenguajes de máquinas diferentes. Con sólo cambiar los microprogramas se cambiaría la máquina que contempla el programador de lenguaje ensamblador.

Sección 7-7

Problemas

1) La unidad de control microprogramada de un computador tiene la estructura que se muestra en la Figura 7.9. Se supone que la memoria de control tiene una longitud de palabra de 24 bits. La parte de control del formato de una microinstrucción está dividida en dos campos. Un campo de control de 13 bits que indica las microoperaciones que se realizan y un campo de selección de dirección que especifica un tipo de bifurcación condicional, que es función de las señales de condición que recibe como entrada la unidad de control. Son posibles 8 señales de condición.

   a) ¿Cuántos bits se necesitan en el campo de selección de dirección?

   b) ¿Cuántos bits tiene el campo de dirección?

   c) ¿Cuál es el tamaño de la memoria de control?

2) Con las mismas especificaciones del problema anterior ¿cómo se puede realizar una ruptura incondicional de secuencia? ¿Cómo se puede evitar la bifurcación?, es decir, describir una microinstrucción que no especifique ningún tipo de bifurcación (condicional o incondicional).

3) A cada una de las instrucciones de un computador se le asignan 8 palabras en la memoria de control de una unidad de control microprogramada. El código de operación de las instrucciones tiene 5 bits y la memoria de control una capacidad de 1024 palabras. Sugerir una transformación de *IR* en *RDC*.

4) La instrucción *JMPNZ* origina una bifurcación si el registro acumulador *AC* es mayor que 0. La dirección de bifurcación se puede expresar de forma directa o indirecta, dependiendo del valor de un bit determinado (*I*) en la instrucción. Sugerir un microprograma que realice esta instrucción para una arquitectura de la unidad de control del tipo de la Figura 7.9.

5) Una CPU tiene 32 registros internos, una ALU con 16 funciones lógicas y 16 funciones aritméticas y un registro de desplazamiento con 8 operaciones posibles. Todos estos elementos se conectan mediante un bus interno de la CPU. Diseñar un formato de microinstrucción que especifique las diferentes microoperaciones.

6) Para el computador pedagógico SIMPLE2 de la sección 7-5 microprogramar las instrucciones máquina del repertorio de instrucciones dado en la Tabla 7.19.

7) Dibujar el esquema de la unidad de control del procesador y obtener el código binario del microprograma correspondiente para la cuarta solución propuesta en el ejemplo 7.3.1.

8) Diseñar una unidad de control microprogramada para el procesador SIMPLE1 del tema 6.

9) Se dispone del repertorio de instrucciones que se muestra en la Tabla 7.23.

Nemotécnico	Instrucción	Significado
LDA x	Cargar acumulador	AC ← M[x]
STA dir	Almacenar acumulador	M[x] ← AC
ADD x	Sumar al acumulador	AC ← AC + M[x]
AND x	AND con el acumulador	AC ← AC AND x
RSH	Desplazamiento lógico a derecha	AC ← AC >> 1
BR x	Salto incondicional a x	PC ← x
BRA x	Salto a x si acumulador = 0	PC ← x si AC = 0

Tabla 7.23: Repertorio de instrucciones

Todas las instrucciones máquina tienen el formato que se muestra en la Figura 7.39: un campo de código de operación (*código_op*) de 3 bits y un campo de dirección de 13 bits.

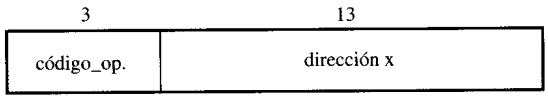


Figura 7.39: Formato de instrucción

Se pide:

a) Diseñar razonadamente un computador capaz de ejecutar este juego de instrucciones, utilizando módulos tales como sumador, registros, buses, memoria, etc.

b) Explicar cómo tienen lugar las operaciones de lectura y escritura en la memoria principal. Suponer que el ciclo básico de reloj de este computador es de 100 ns y que tanto el ciclo de lectura como de escritura de la memoria principal duran 300 ns.

c) Diseñar una unidad de control microprogramada para este computador. Discutir los posibles formatos de microinstrucciones (μI). Suponer que la memoria de control dispone de 256 palabras de un ancho suficiente.

Justificar todas las suposiciones que deban realizarse.

10) Para el computador diseñado en el problema anterior, se pide:

a) Escribir el microcódigo correspondiente a cada una de las instrucciones máquina de su repertorio.

b) Escibir los microprogramas encargados de la búsqueda y decodificación de las instrucciones máquina.

c) Dar el contenido de la memoria de control.

11) Dado un computador con 7 líneas de petición de interrupción, utilícese un codificador de prioridad convencional de 8 entradas y 3 salidas para realizar la identificación y tratamiento a nivel de microprograma de las peticiones de interrupción. Especifíquense los contenidos que haya que cargar en la memoria de control.

12) Dado un computador microprogramado con la estructura que se muestra en la Figura 7.40 y dados:

RSE (retardo del secuenciador) = 50 ns

RPR (retardo de la unidad de procesamiento) = 90 ns

TAM (tiempo de acceso a memoria de control) = 65 ns

SPR (tiempo de preparación y propagación de señales en cada uno de los registros) = 15 ns

se pide:

a) Explicar detalladamente cómo se ejecuta una microinstrucción (μI) en este computador.

b) Explicar detalladamente los factores que determinan si el salto se produce o no en una microinstrucción de salto condicional.

c) Determinar el tiempo de ciclo mínimo de la máquina.

d) Comparar el tiempo obtenido en el apartado anterior con el correspondiente al caso en que no hay solapamiento de búsqueda y ejecución de microinstrucciones.

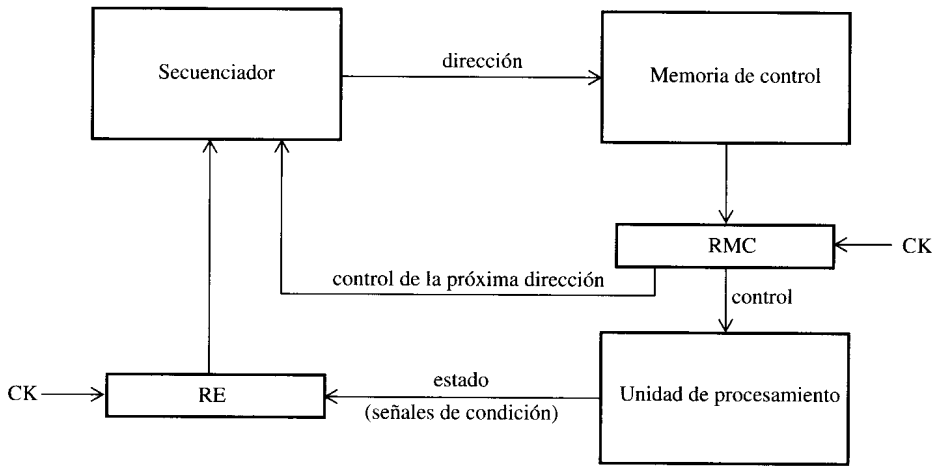


Figura 7.40: Estructura del computador

Se supone que el bloque secuenciador contiene la lógica para el cálculo de la dirección de la próxima microinstrucción.

13) Dado un computador microprogramado con la estructura que se muestra en la Figura 7.41 y tomando como válidos los tiempos del problema anterior, se pide:

a) Explicar detalladamente cómo se ejecuta una microinstrucción en este computador.

b) Explicar detalladamente los factores que determinan si el salto se produce o no en una microinstrucción de salto condicional.

c) Determinar el tiempo de ciclo mínimo de la máquina.

d) Suponer que para el diseño se puede optar entre varias memorias de control con distintos tiempos de acceso. Hallar para qué valores de *TAM* el tiempo de ciclo de la máquina obtenido en el problema anterior resuelta menor que el obtenido con este nuevo diseño.

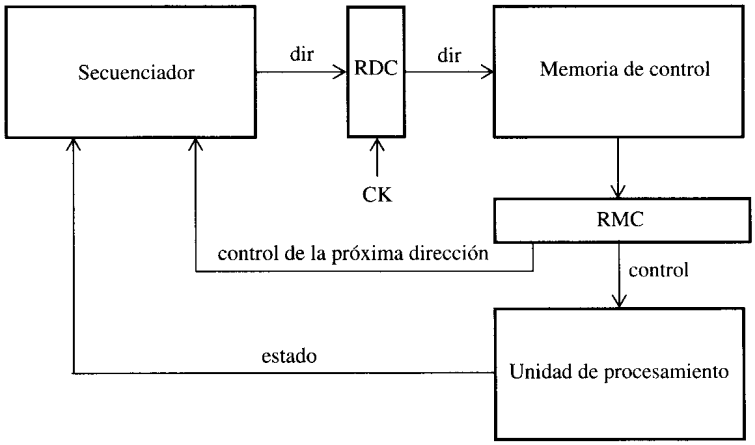


Figura 7.41: Estructura del computador

14) Repetir el problema anterior considerando un computador con la estructura que se da en la Figura 7.42.

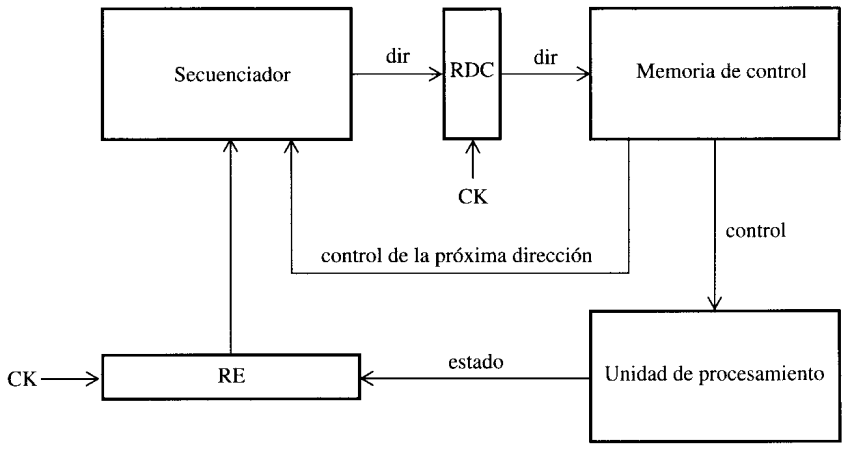


Figura 7.42: Estructura del computador

Apéndice A

Componentes combinacionales

Un *circuito combinacional* es un circuito lógico cuyos valores de salida están determinados en cualquier instante de tiempo únicamente por los valores aplicados a sus entradas, y por tanto son independientes de los estados anteriores de las mismas. Los circuitos combinacionales no nos permiten pues almacenar el estado de las entradas y utilizarlas posteriormente para tomar decisiones, es decir, son circuitos que no tienen memoria.

La característica principal de un circuito combinacional es la función lógica que realiza. Esta función lógica describe el comportamiento del circuito pero no tiene por qué describir su estructura interna, por lo que es posible que existan diferentes circuitos lógicos que realizan la misma función. Los circuitos combinacionales tienen dos usos principales en los sistemas digitales:

a) *Transferencia de datos*. Controlan el flujo de señales lógicas de una parte del sistema a otra.

b) *Procesamiento de datos*. Procesan o transforman los datos realizando los cálculos necesarios.

En este apéndice se consideran los principales componentes combinacionales que se emplean en la descripción a nivel de registro de un sistema digital.

1) Puertas de palabras
2) Codificadores
3) Decodificadores
4) Multiplexores
5) Demultiplexores
6) Dispositivos lógicos programables

Aunque algunos de estos componentes ya se han introducido en los propios temas (por ejemplo los decodificadores) se expone a continuación brevemente, y con la finalidad de hacer el texto lo más autocontenido posible, la función lógica que realiza cada uno de ellos. No se consideran los circuitos sumadores, multiplicadores, comparadores y unidades aritmético-lógicas ya que a su estudio exclusivo se dedicó el tema 4.

Sección A-1

Puertas de palabras

Se utilizan para representar la realización de una función lógica, bit a bit, entre dos palabras o bien entre una palabra y un bit constante.

El símbolo de la puerta de palabra expresa la operación que se efectúa entre sus datos de entrada y la correspondencia entre símbolos y funciones lógicas es la misma que se utiliza con las puertas lógicas básicas, con la única diferencia de explicitar en las entradas y salidas el número de bits que contiene.

Así, dados $X = (x_{n-1},\ldots, x_1, x_0)$ e $Y = (y_{n-1},\ldots, y_1, y_0)$, en la Figura A.1 se muestran respectivamente las puertas AND, OR y OR exclusiva de palabras de n bits.

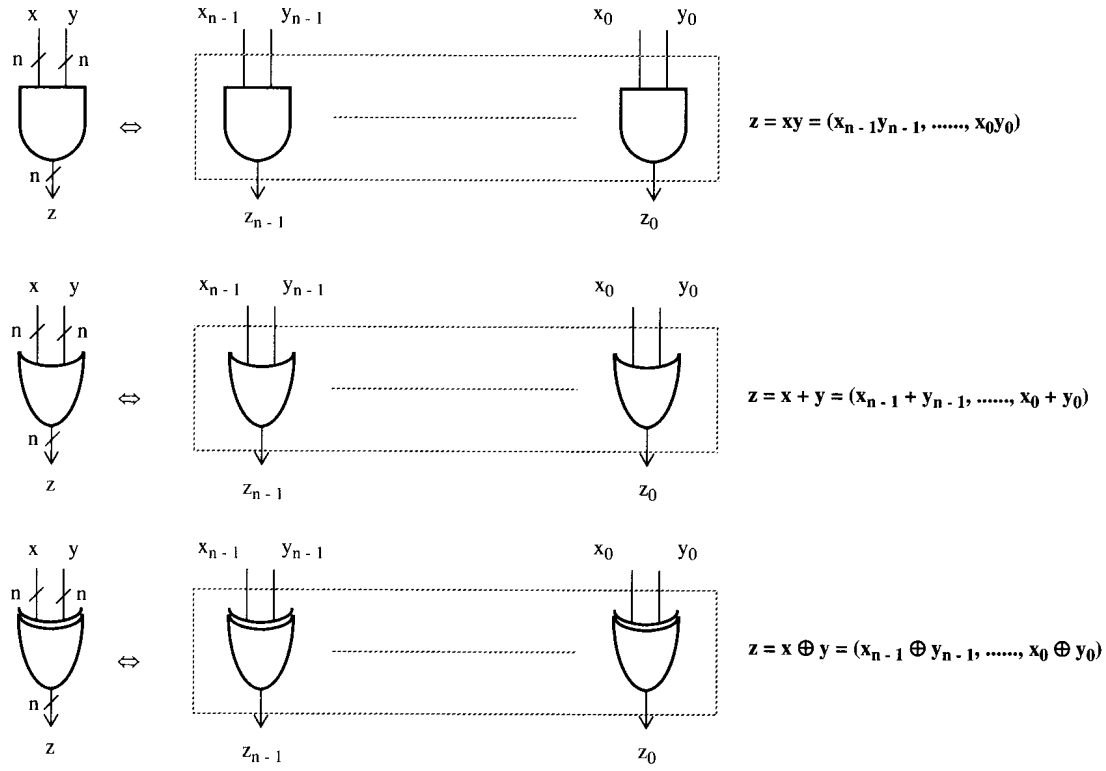


Figura A.1: Puertas AND, OR y OR exclusiva de palabras de n bits

Sección A-2

Codificadores

Los codificadores binarios son circuitos combinacionales que poseen 2^n entradas $y = (y_{2^n-1}, ..., y_1, y_0)$ y n salidas $z = (z_{n-1}, ..., z_1, z_0)$ tal como se muestra en la Figura A.2. Realizan la función inversa de los decodificadores y se pueden considerar como un conversor de código que transforma un código del tipo 1 entre 2^n a un código binario. Es decir, en cualquier instante de tiempo sólo una de las variables de entrada tiene valor 1 y las salidas representan, en un código binario, el índice de la entrada con valor 1. Si hay más de una entrada con el valor 1, la salida está indefinida. Un ejemplo de utilización es en los teclados donde al pulsar una tecla se activa una de las entradas de un codificador que se encarga de codificar la tecla pulsada.

El módulo también tiene una entrada H de habilitación, de forma que cuando $H = 0$ todas las salidas del decodificador valen 0 y una salida A que si vale 1 quiere decir que el codificador está activo. Esta salida indica que hay una entrada con valor 1 ("activa"). La entrada H y la salida A se emplean cuando se desea diseñar una red de módulos codificadores con un mayor número de entradas o generar una interrupción.

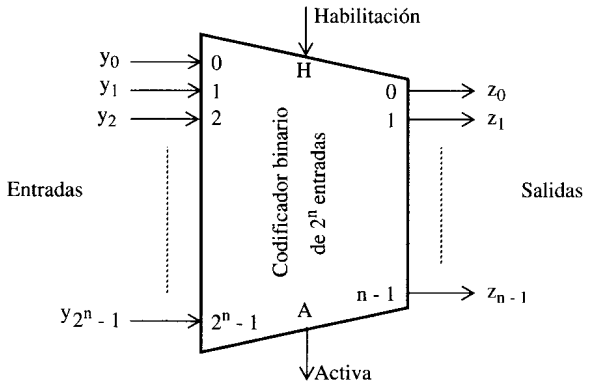


Figura A.2: Diagrama funcional de un codificador binario de 2^n entradas

Una descripción de alto nivel de un codificador binario de 2^n entradas es:

$z = i$ si $y_i = 1$ e $y_k = 0 \; \forall k \neq i$ y $H = 1$

$z = 0$ en cualquier otro caso

$A = 1$ si algún $y_i = 1$ y $H = 1$

$A = 0$ en cualquier otro caso

donde $z = \sum_{j=0}^{n-1} z_j 2^j$, y tal que $0 \leq i, k \leq 2^n - 1$.

Las expresiones lógicas que representan un codificador binario son:

$$z_i = H \sum y_k$$

donde y_k se incluye en la suma si el bit i-ésimo de la representación binaria de k es 1, y

$$A = H \sum_{i=0}^{2^n - 1} y_i$$

Por ejemplo, las expresiones lógicas que representan un codificador binario de 4 entradas son:

$$z_0 = H(y_1 + y_3)$$
$$z_1 = H(y_2 + y_3)$$
$$A = H(y_0 + y_1 + y_2 + y_3)$$

En la Figura A.3 se muestra la realización con puertas lógicas de un codificador binario de 4 entradas.

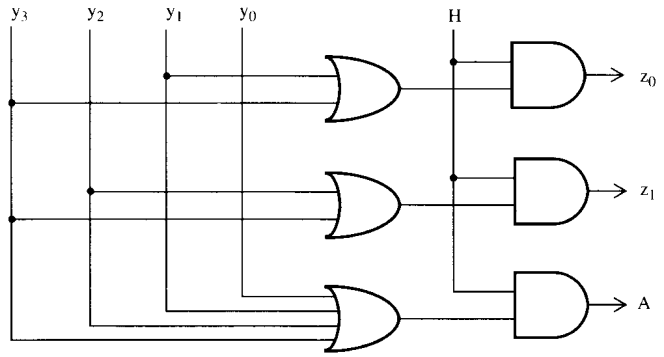


Figura A.3: Realización de un codificador binario de 4 entradas

Los codificadores se utilizan siempre que se desea identificar la ocurrencia de un suceso de un conjunto de posibles sucesos disjuntos, es decir, que no se pueden presentar simultáneamente. Cada suceso se identifica unívocamente mediante un número entero. Hay codificadores para otros códigos distintos del binario (BCD, Gray, etc).

A.2.1 Codificadores con prioridad

Los codificadores que se han considerado anteriormente presentan la limitación de que en cualquier instante de tiempo solamente una entrada puede tener el valor *1*. Como no hay nada que impida el que se activen

simultáneamente varias líneas de entrada de un codificador, los circuitos codificadores se diseñan normalmente para que respondan a una sola señal de entrada activa. Esto se hace asignando prioridades a las entradas, de forma que en cada instante sólo se genera el código correspondiente a la entrada activa que tenga la mayor prioridad, se habla entonces de *codificadores con prioridad*. El orden de la prioridad está fijado en el codificador y para un codificador de 2^n entradas se suele asignar a y_0 la menor prioridad y a y_{2^n-1} la mayor.

Para diseñar codificadores con prioridad de un número de entradas elevado se necesitan una entrada de habilitación H_e y dos salidas H_s y A. Una descripción de alto nivel de un codificador con prioridad de 2^n entradas es:

$z = i$ si $y_i = 1$ e $y_k = 0 \; \forall \, k \neq i$ y $H_e = 1$

$z = 0$ en cualquier otro caso

donde $z = \sum_{j=0}^{n-1} z_j 2^j$, y tal que $0 \leq i, \; k \leq 2^n - 1$.

$H_s = i$ si $\forall \, i$ $y_i = 1$ y $H_e = 1$

$H_s = 0$ en cualquier otro caso

y

$A = 1$ si $\exists$ al menos un $y_i = 1$ y $H_e = 1$

$A = 0$ en cualquier otro caso

Los codificadores con prioridad se utilizan para seleccionar un suceso de un conjunto que pueden ocurrir simultáneamente. A este suceso lo representa por un número entero que lo codifica de forma unívoca. En el apartado 3.4.5 del tema 3 en el que se trata la *identificación de la fuente de interrupción mediante hardware paralelo* se presentó una aplicación del codificador con prioridad (ver Figura 3.21). Tal como se muestra en la Figura A.4, un codificador con prioridad se puede realizar empleando dos subsistemas. El primer subsistema corresponde a un circuito de resolución de la prioridad, cuyo objetivo es eliminar todos los 1's excepto el de mayor prioridad; el segundo es un codificador binario del tipo visto en el apartado anterior.

El circuito de resolución de prioridad tiene 2^n entradas (x_{2^n-1}, ..., x_0) y 2^n salidas (y_{2^n-1}, ..., y_0). Su descripción de alto nivel es:

$y_i = i$ si $x_i = 1$ y $x_k = 0, \; \forall \, k > i$

$y_i = 0$ en cualquier otro caso

Esto corresponde a la siguiente expresión lógica:

$$y_i = \overline{x}_{2^n-1} \; \ldots\ldots \; \overline{x}_{i+1} \; x_i$$

donde $\overline{x}_j$ representa el complemento de x_j. En la Figura A.5 se muestra la realización mediante puertas AND de un circuito de 4 entradas para la resolución de prioridad.

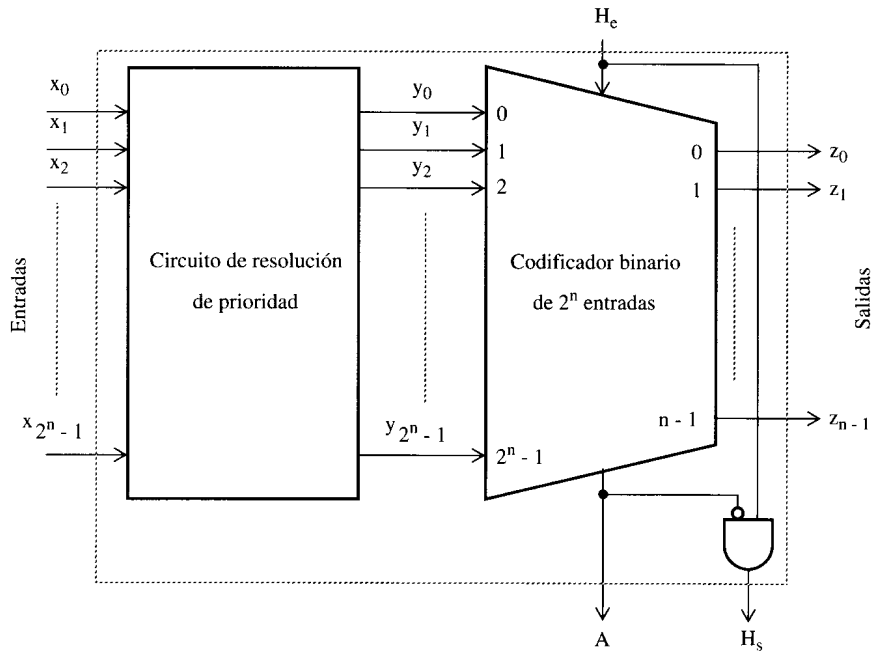


Figura A.4: Diagrama de bloques de un codificador con prioridad

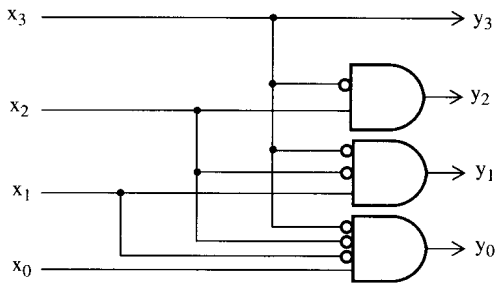
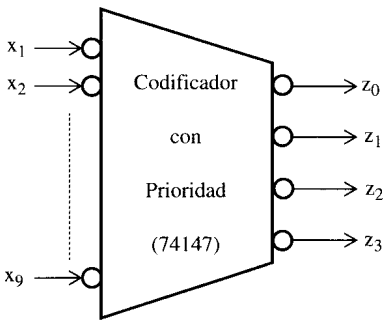


Figura A.5: Circuito paralelo de 4 entradas de resolución de prioridad

En la Figura A.6 se muestra el circuito integrado *74147* y su tabla de verdad, se trata de un codificador con prioridad, con 9 entradas (x_1, x_2, ..., x_9) y 4 salidas (z_0, z_1, z_2, z_3). El símbolo × corresponde a que el valor de la entrada puede ser 0 ó 1.

Las entradas son activas en baja y con la prioridad creciente con el número de entrada, es decir la entrada x_9 tiene la mayor prioridad. El código de salida corresponde al BCD con complementación en todas las salidas, se trata por tanto del circuito integrado inverso al *7445* (decodificador de BCD a decimal).



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	z_3	z_2	z_1	z_0
1	1	1	1	1	1	1	1	1	1	1	1	1
×	×	×	×	×	×	×	×	0	0	1	1	0
×	×	×	×	×	×	×	0	1	0	1	1	1
×	×	×	×	×	×	0	1	1	1	0	0	0
×	×	×	×	×	0	1	1	1	1	0	0	1
×	×	×	×	0	1	1	1	1	1	0	1	0
×	×	×	0	1	1	1	1	1	1	0	1	1
×	×	0	1	1	1	1	1	1	1	1	0	0
×	0	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0

Figura A.6: Diagrama de bloques y tabla de verdad del codificador 74147

Cuando ninguna entrada está activada el codificador *74147* lo interpreta como que la décima entrada, la entrada x_0 de menor prioridad, está activada y todas las salidas se ponen en alta, que corresponde al complemento del código BCD "0000". En la Figura A.7 se muestra la realización mediante puertas lógicas del circuito integrado *74147*.

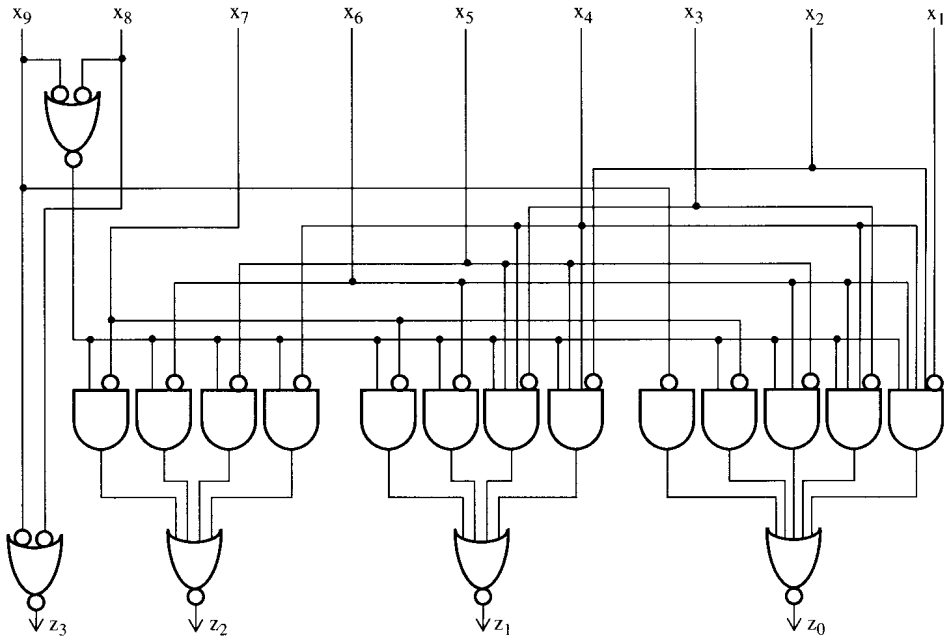


Figura A.7: Realización con puertas lógicas del codificador 74147

Sección A-3

Decodificadores

Los circuitos decodificadores son módulos combinacionales que *transforman los datos de un código numérico determinado (código de n bits) al código con un formato conocido como* 1 *entre* 2^n.

Estos módulos se utilizan fundamentalmente en el direccionamiento de los circuitos de memorias y de dispositivos de E/S. En esta aplicación el objetivo es seleccionar una palabra de entre 2^n palabras de memoria, o un dispositivo de entre 2^n dispositivos de E/S en operaciones de lectura o escritura. Para ello cada salida del decodificador se debe conectar a la entrada de control correspondiente de cada dispositivo o posición de memoria. En general un *decodificador binario de n entradas* (ver Figura A.8) es un sistema combinacional que posee n entradas $y = (y_{n-1}, ..., y_1, y_0)$ y 2^n salidas $z = (z_{2^n-1}, ..., z_1, z_0)$.

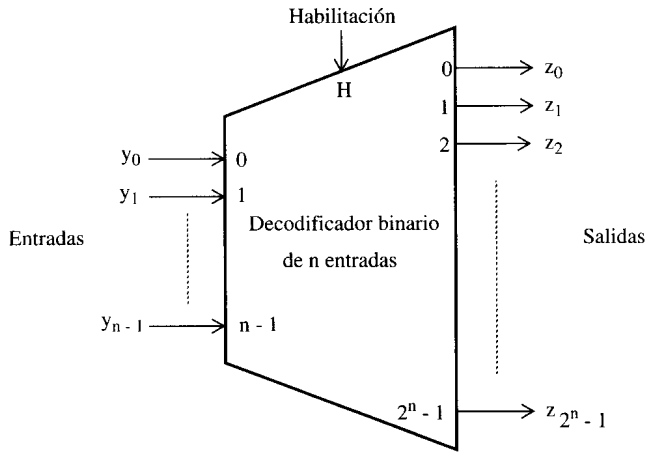


Figura A.8: Diagrama funcional de un decodificador binario de n entradas

El vector de entrada y se puede considerar que representa un nº entero comprendido entre 0 y 2^n - 1. Cuando la entrada es el número i, la salida y_i vale 1 y todas las otras salidas están a 0.

Para faciltar su uso en el diseño de circuitos el decodificador incorpora una entrada de habilitación del módulo H. Cuando H = 0 todas las salidas del decodificador valen 0. Una descripción de alto nivel de un decodificador binario de n entradas es:

$$z_i = 1 \quad \text{si} \quad y = i \quad \text{y} \quad H = 1$$
$$z_i = 0 \quad \text{en cualquier otro caso}$$

donde $y = \sum_{j=0}^{n-1} y_j 2^j$.

La expresión lógica que representa un decodificador binario es:

$$z_i = H\, m_i(y) \quad i = 0, 1, ..., 2^n - 1$$

donde $m_i(y)$ es el minterm i-ésimo de las n variables y.

En la Figura A.9 se da la realización con puertas lógicas de un decodificador binario de 2 entradas.

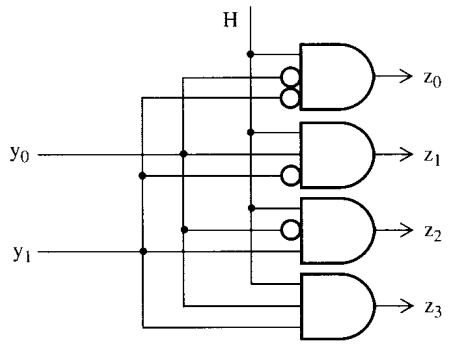


Figura A.9: Realización de un decodificador binario de 2 entradas

Los decodificadores se utilizan siempre que un conjunto de valores que se han codificado previamente tienen posteriormente que separarse (es decir decodificarse). Un ejemplo es el código de operación de un computador. Este código, que forma parte de cada instrucción, se tiene que decodificar para determinar la operación que el computador tiene que ejecutar (ver Figura A.10).

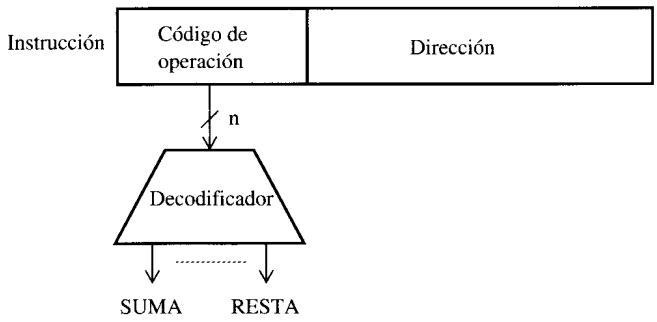


Figura A.10: Decodificación de instrucciones

En la Figura A.11 se muestra un decodificador de 1 entre 10 (se trata del circuito integrado *7445* que es un decodificador de código BCD a decimal). Tal como se indica mediante los círculos inversores, las señales de salida de este decodificador son activas a nivel bajo (lógica negativa). Así, si la entrada al decodificador es $y_3y_2y_1y_0 = 0011$ (que es en binario el número 3) su salida será $z_9z_8z_7z_6z_5z_4z_3z_2z_1z_0 = 1111110111$. El 0 en la línea de salida z_3 indica que se ha decodificado la entrada que corresponde al número 3.

Este circuito no tiene línea de habilitación, sin embargo resulta inhabilitado para cualquiera de las seis combinaciones de entradas que no representan un dígito BCD válido.

En la Tabla A.1 se da la tabla de verdad del circuito *7445*. Se observa que para las configuraciones de entrada que corresponden a los números decimales que van del 10 al 15 todas las salidas del decodificador están a 0 indicando así que no está habilitado.

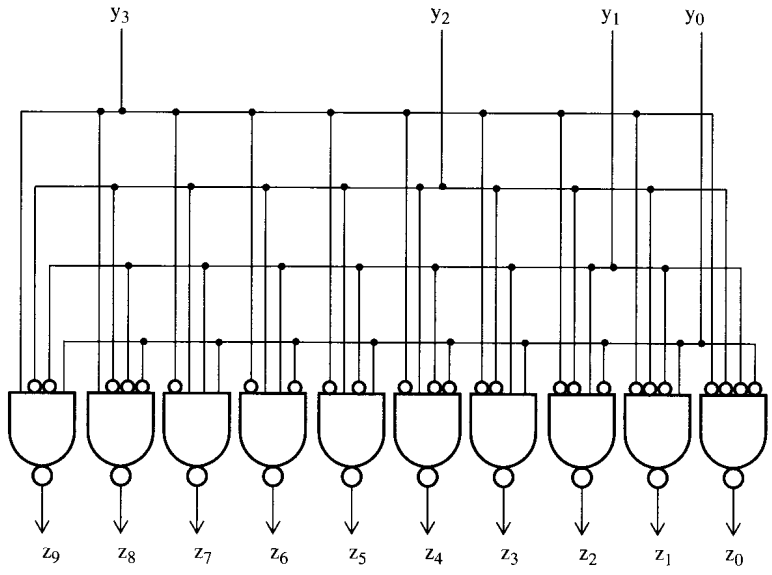


Figura A.11: Circuito lógico del decodificador 7445

Un decodificador de n entradas y una puerta OR realiza cualquier función de n variables. La salida i-ésima del decodificador corresponde al minterm $m_i(y)$, esto permite obtener la expresión canónica de la función como suma de minterms, conectando directamente a una puerta OR las salidas del decodificador que corresponden a los minterms de la función que se desea sintetizar. Con el mismo decodificador se pueden generar más de una función de las mismas variables.

Sean por ejemplo las siguientes funciones lógicas de tres variables expresadas en forma canónica como suma de sus minterms:

$$f_1(y_3, y_2, y_1) = \sum m(0, 3, 4)$$

$$f_2(y_3, y_2, y_1) = \sum m(1, 2)$$

$$f_3(y_3, y_2, y_1) = \sum m(3, 5, 7)$$

y_3	y_2	y_1	y_0	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Tabla A.1: Tabla de verdad del decodificador 7445

En la Figura A.12 se muestra la síntesis de estas funciones utilizando un decodificador binario de 3 entradas y 3 puertas OR.

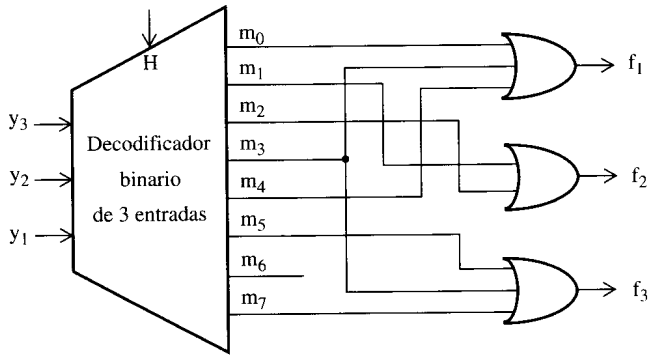


Figura A.12: Síntesis de funciones lógicas utilizando un decodificador

Los decodificadores se pueden interconectar entre sí para obtener módulos de mayor número de líneas de entrada y de salida (propiedad de ampliación de los módulos). La manera de realizarlo es mediante una estructura arborescente de decodificadores, tal como se muestra en la Figura A-11 en el caso de un decodificador de 4 entradas construido a partir de decodificadores de 2 entradas.

El árbol, de dos niveles, tiene un decodificador en el primer nivel y cuatro en el segundo. El vector de entrada $y = (y_3, y_2, y_1, y_0)$ se divide en dos subvectores, con (y_3, y_2) decodificados en el primer nivel e (y_1, y_0) en el segundo. Las 16 salidas se particionan en 4 grupos con 4 salidas en cada uno. La decodificación del subvector (y_3, y_2) habilita a uno de los grupos y la decodificación de (y_1, y_0) produce la salida correspondiente en el grupo que está activado.

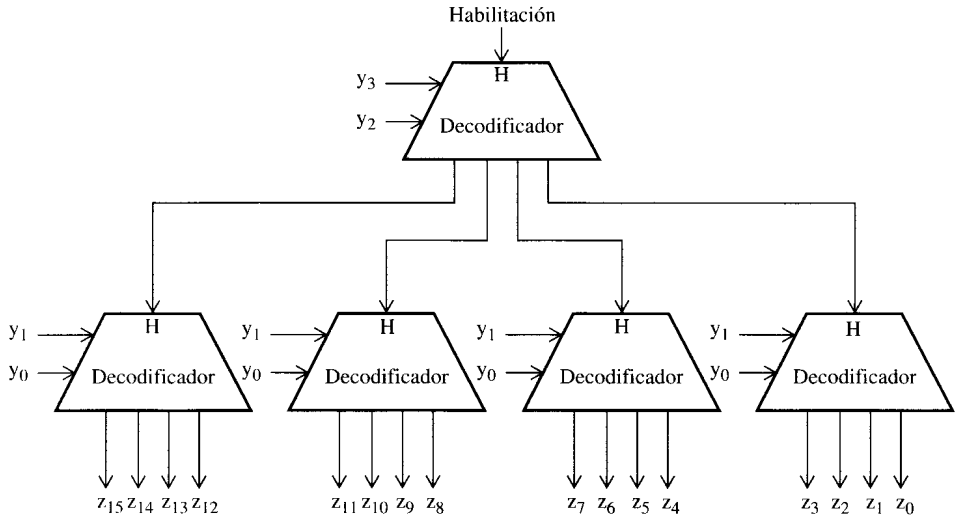


Figura A.13: Decodificador de 4 entradas obtenido mediante decodificadores de 2 entradas

Sección A-4

Multiplexores

La función primaria de un multiplexor (que en ocasiones se abrevia MUX) como componente combinacional en un computador es *hacer que palabras con orígenes diferentes se puedan transferir a un mismo destino.* Permite seleccionar una de entre varias señales de entrada y enviarla a la salida, por esta razón a un multiplexor también se le conoce como *circuito selector de datos.*

En un instante dado, sólo se transfiere una palabra a través del multiplexor, es decir, el bus de salida se comparte por todas las fuentes de datos. A esta manera de compartir se la conoce como *multiplexado en el tiempo.* El número de *líneas de selección o control* es tal que sus posibles combinaciones igualan al número de *entradas de datos*, es decir, si el circuito multiplexor admite 8 líneas de entrada, debe tener 3 líneas de selección, pues $2^3 = 8$ son todas sus posibles combinaciones. En general, para n líneas de selección o control (c_i, $i = 0, 1, ..., n\text{-}1$) hay disponibles hasta 2^n líneas de entradas de datos (y_i, $i = 0, 1, ..., 2^n - 1$).

El multiplexor dispone de una *línea de habilitación* (*H*) que cuando está desactivada (a 0 ó a 1 según se trate), la salida del multiplexor *z* toma un valor fijo independientemente del valor de las entradas de selección y de datos (ver Figura A.14). En algunos casos si el multiplexor no está habilitado su salida se encuentra en un estado de alta impedancia (ver apartado 1.4.5).

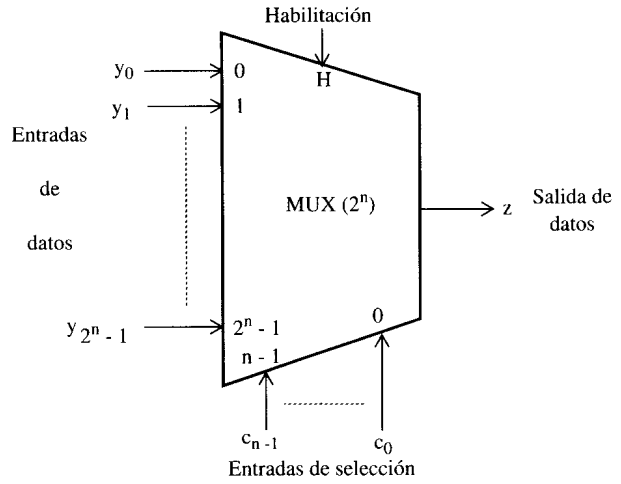


Figura A.14: Diagrama funcional de un multiplexor de 2^n entradas de datos

Los valores de las variables de control c se interpretan como la representación binaria de un número entero comprendido entre 0 y 2^n -1. La salida z del multiplexor corresponde al dato de entrada y_i si c representa el entero i. Una descripción de alto nivel del multiplexor es:

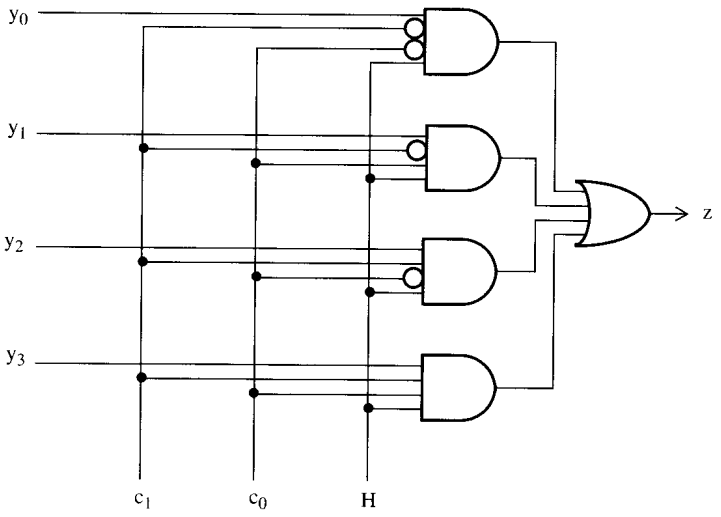
$z = y_c$	si el MUX está habilitado	H = 1
$z = 0$	si el MUX está deshabilitado	H = 0

donde

$$c = \sum_{j=0}^{n-1} c_j 2^j$$

En la Figura A.15 se muestra la realización mediante inversores, puertas AND y una puerta OR de un multiplexor con 4 entradas de datos de un bit cada una (y_3, y_2, y_1 e y_0) y que por tanto requiere un bus de control de 2 bits (c_1, c_0). También se indica la tabla de verdad del multiplexor.

Así si el multiplexor está habilitado (H = 1) y las entradas de selección o control valen $c_1 c_0$ = 0 1 (que es la representación binaria del número entero c = 1), de acuerdo con la descripción de alto nivel del multiplexor su salida será $z = y_1$.



c_1	c_0	H	z
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	y_0
0	1	1	y_1
1	0	1	y_2
1	1	1	y_3

Figura A.15: Realización de un multiplexor de 4 entradas de datos

El multiplexor, si no se tienen en cuenta los inversores, es un ejemplo típico de circuito combinacional de dos niveles (AND-OR). Siempre se puede obtener un multiplexor más grande (mayor número de entradas de datos) interconectando adecuadamente varios multiplexores más pequeños (propiedad de ampliación de los módulos). Por ejemplo, en la Figura A.16 se presenta el diseño de un multiplexor de 4 entradas de datos (MUX(4)) utilizando 3 multiplexores de 2 entradas de datos (MUX(2)).

Esta propiedad de construcción mediante una estructura arborescente de multiplexores de más entradas, a partir de multiplexores con menor número de entradas (análoga al caso de los decodificadores) es totalmente general.

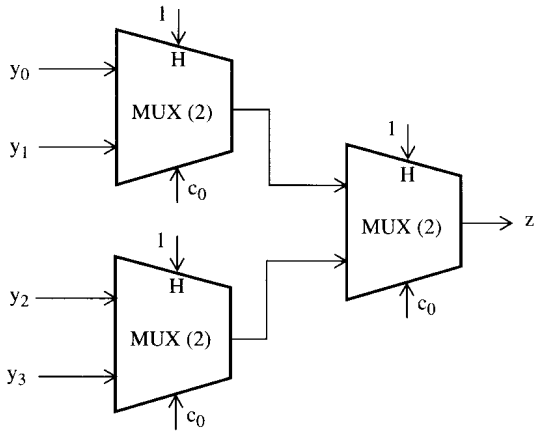


Figura A.16: MUX (4) obtenido mediante 3 MUX (2)

Además de su utilidad para seleccionar o transferir datos, otra característica muy interesante de los multiplexores es que pueden servir también como *módulos universales para la generación de funciones lógicas*. En el caso más simple, se elige un multiplexor de tantas líneas de selección como variables tenga la función lógica. Cada variable se conecta a una línea de selección y las entradas de datos del multiplexor se ponen a un valor fijo (0 ó 1) correspondiente a cada uno de los valores de la tabla de verdad de la función lógica que se desea generar. Por ejemplo, en la Figura A.17 se muestra la realización de la función OR-exclusiva de tres variables con un multiplexor de 8 entradas de datos.

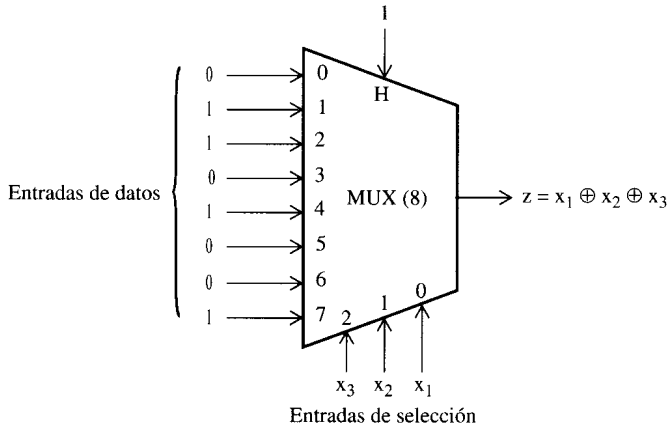


Figura A.17: Realización de la función $x_1 \oplus x_2 \oplus x_3$ con un MUX (8)

Sección A-5

Demultiplexores

Un demultiplexor es un circuito combinacional, con n entradas de selección o control $c = (c_{n-1}, ..., c_1, c_0)$, una entrada de datos y, una entrada de habilitación del módulo H y 2^n salidas de datos $z = (z_{2^n-1}, ..., z_1, z_0)$ (ver Figura A.18). La entrada se conecta con la salida seleccionada por las variables de control ó selección. Todas las otras salidas están a 0.

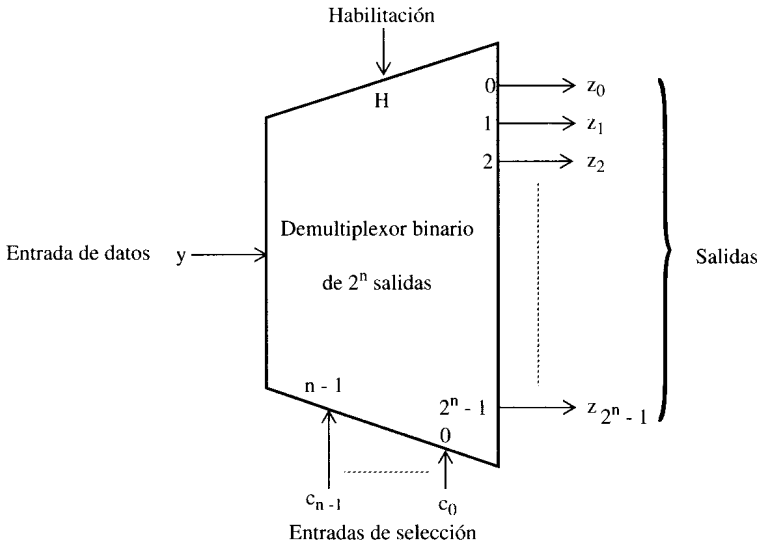


Figura A.18: Diagrama funcional de un demultiplexor de n entradas y 2^n salidas

Los demultiplexores se pueden emplear para transferir datos desde una fuente común a uno de entre varios posibles destinos bajo el control de las entradas de selección. Por este motivo se suele denominar también a los demultiplexores como distribuidores. Esta operación es la inversa de la de conducción de datos realizada por un multiplexor.

Una descripción de alto nivel de un demultiplexor de n entradas es:

$$z_i = y \quad si \quad i = c \quad y \quad H = 1$$
$$z_i = 0 \quad si \quad i \neq c \quad o \quad H = 0$$

donde $c = \sum_{j=0}^{n-1} c_j 2^j$, y tal que $0 \leq i \leq 2^n - 1$.

La expresión lógica que representa las salidas del demultiplexor es:

$$z_i = H \text{ y } m_i(c) \qquad i = 0, 1, ..., 2^n - 1$$

donde $m_i(c)$ es el minterm i-ésimo de las n variables de selección c. Estas expresiones son similares a las del decodificador con la única diferencia de la entrada de datos adicional y. En consecuencia, un demultiplexor se puede utilizar como un decodificador. Es posible también emplear de forma inversa un decodificador como un demultiplexor, a condición de que la entrada de habilitación H del decodificador se use como entrada de datos del demultiplexor. En este último caso el demultiplexor resultante no dispondría de entrada de habilitación.

Sección A-6

Dispositivos lógicos programables

Existen elementos lógicos de propósito general que han sido desarrollados para realizar cualquier función combinacional, como por ejemplo determinadas partes de la unidad de control de un computador no microprogramable. En general estos componentes han de cumplir los dos requisitos siguientes:

1) Cada componente debe tener una estructura subyacente uniforme, que permita una producción masiva utilizando las modernas técnicas de fabricación de circuitos integrados.

2) Es posible alterar la estructura básica del componente, o bien durante el propio proceso de fabricación o directamente por parte del usuario, para realizar la función lógica impuesta por el diseño.

Ejemplos de este tipo de circuitos son:

a) ROM (Read Only Memory).

b) PLA (Programmable Logic Array).

c) PAL (Programmable Array Logic).

La utilización de la ROM como elemento de propósito general para la síntesis de funciones lógicas ya se comentó en el tema 4 (ver apartado 4.2.6 y problema 4.19) y por ese motivo no se considera. Simplemente conviene recordar que con una ROM de 2^n palabras $\times$ m bits se pueden realizar simultáneamente m funciones lógicas de n variables.

A.6.1 Arrays lógicos programables (PLA)

Un *array lógico programable* o PLA (Programmable Logic Array) es un conjunto de puertas lógicas sin un propósito definido a priori, que se puede configurar para la realización de una determinada función. Esta particularización se lleva a cabo normalmente mediante la fusión de determinados fusibles que interconectan las puertas lógicas del PLA. Un PLA contiene los elementos siguientes:

a) Un conjunto de líneas de entrada (del orden de 8-16)

b) Un conjunto de líneas de salida (del orden de 6-8)

c) Un conjunto de puertas AND (del orden de 48-96)

d) Un conjunto de puertas OR (una por cada línea de salida)

e) Un conjunto de inversores sobre cada línea de entrada y salida

f) Dos matrices de conexiones alterables (matriz AND y matriz OR)

En la Figura A.19 se muestra un diagrama de bloques de la estructura de un PLA.

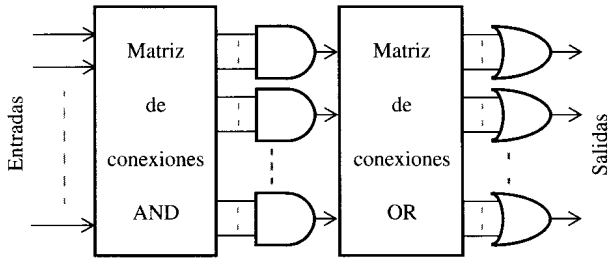


Figura A.19: Estructura funcional de un PLA

A nivel lógico un PLA permite una realización NOT-AND-OR de un conjunto de funciones. Específicamente un PLA $n \times m$ permite sintetizar m funciones de n variables. Consta de un conjunto de n *inversores* que proporcionan el valor complementario de las variables de entrada, un conjunto de k *puertas AND* que producen los términos productos y un conjunto de m *puertas OR* que generan los términos sumas.

Las puertas AND y OR se organizan en dos estructuras regulares denominadas arrays AND y OR respectivamente. Las conexiones de las líneas de entrada (o de sus complementos) con las entradas de las puertas AND se programan. Análogamente, también se programan las conexiones entre las salidas de las puertas AND y las entradas a las puertas OR. En la Figura A.20 se ilustra la estructura lógica de un PLA. Se observa que un PLA $n \times m$ que contiene k puertas AND puede realizar un conjunto de m funciones de conmutación de n variables, si el número de términos productos en su representación de suma de productos no es mayor que k.

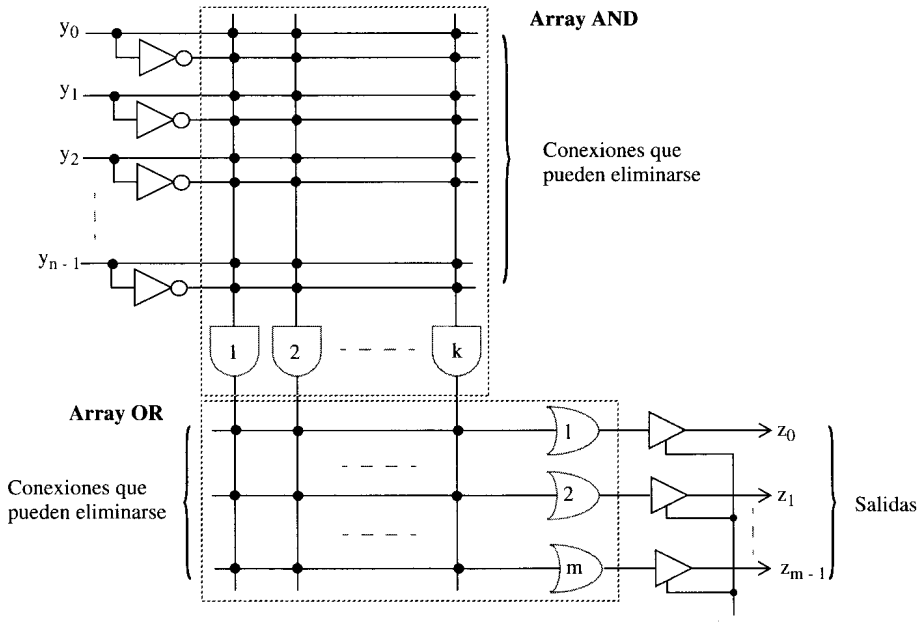


Figura A.20: Estructura lógica de un PLA

En la Figura A.21 se muestran las cuatro condiciones que hay para cada variable de entrada.

1) El estado inicial no programado con todas las conexiones intactas tanto de la variable como de su complemento.
2) La variable o su complemento se seleccionan eliminando la conexión correspondiente.
3) Si se eliminan ambas conexiones la variable no aparece en el término producto.
4) Si se eliminan todas las conexiones de una puerta AND, se genera a su salida un 1 lógico.

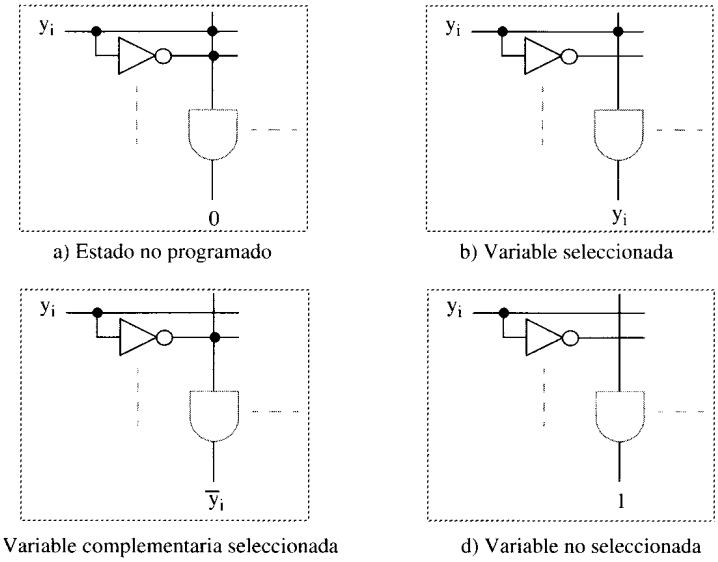


Figura A.21: Estados de conexión en el array AND de un PLA

En general no se debería utilizar la condición de estado no programado ya que esto llevaría a que de forma permanente el término producto sería 0 ($py_i\overline{y}_i = 0$, donde p representa las otras variables que forman el término producto). Una excepción a esta regla es cuando no se desea utilizar una puerta AND en cuyo caso hay que emplear el estado no programado que deja intacta todas las conexiones y hace que el término producto correspondiente sea 0.

Conviene señalar las diferencias, que desde el punto de vista de realización de funciones lógicas presentan una ROM y un PLA.

1) En general un PLA produce una síntesis más compacta que una ROM, ya que en lugar de almacenar la tabla de verdad completa de la función, realiza una suma de productos mínima. La reducción en tamaño resulta considerable si el número de productos que se necesitan es mucho menor que el máximo de 2^n.
2) La realización con un PLA está limitada a un conjunto de funciones, que pueden representarse por un conjunto de sumas de productos que no tiene mas de k términos productos. Esta restricción no existe en el caso de la ROM que incorpora todos minterms de la función.

3) La programación de un PLA es más difícil que el de una ROM. Esto se debe a que en un PLA hay que diseñar una estructura AND-OR de dos niveles.

A.6.2 Ejemplo: Síntesis de funciones lógicas con un PLA

Se desea implementar con un PLA el siguiente conjunto de funciones booleanas:

$$f_1(y_3, y_2, y_1, y_0) = y_0\overline{y}_1y_2$$

$$f_2(y_3, y_2, y_1, y_0) = y_0y_2 + y_1y_2$$

$$f_3(y_3, y_2, y_1, y_0) = \overline{y}_0\overline{y}_1 + y_0\overline{y}_2\overline{y}_3 + y_0\overline{y}_1y_2$$

En la Figura A.22 se muestran las conexiones de un PLA para estas funciones. Conviene observar que cuando existe un término producto que es común a varias funciones se puede compartir de la misma forma que en una implementación normal AND-OR en dos niveles (la salida de la puerta AND número 5 forma el término producto $y_0\overline{y}_1y_2$ que es compartido por las funciones f_1 y f_3). La simplificación lógica puede ser útil también con una implementación PLA debido al número limitado de puertas que se dispone.

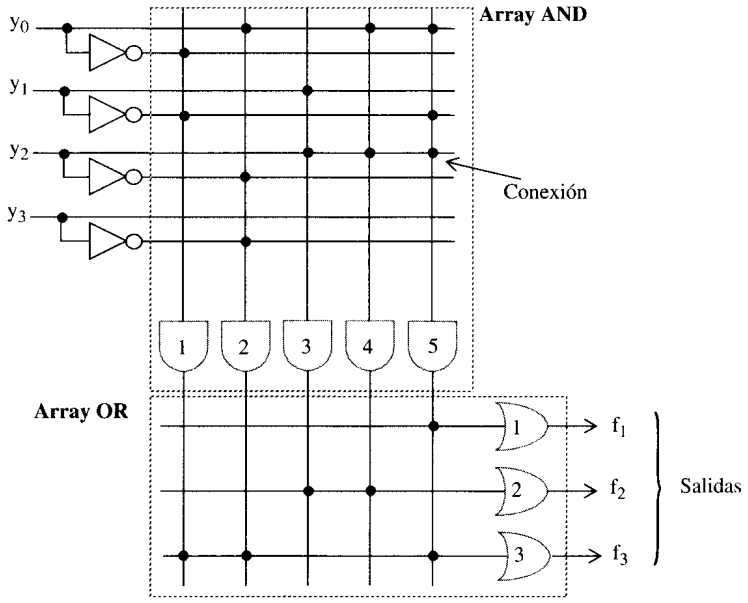


Figura A.22: Síntesis de las funciones f_1, f_2 y f_3 del ejemplo con un PLA

A.6.3 Arrays lógicos programables (PAL)

El PLA descrito en la sección anterior tiene programable tanto las conexiones del array AND como del array OR. La compañía Advanced Micro Devices (AMD) fabrica una familia de dispositivos lógicos programables que solo permita alterar las conexiones del array AND. Denominan a este tipo de módulo como PAL (Programmable Array Logic), término que es ahora ampliamente aceptado para nombrar a esta clase de dispositivos.

En un PAL la salida de cada puerta AND va directamente a la entrada de una puerta OR. Típicamente, se conectan 8 puertas AND a las entradas de cada puerta OR (de 8 entradas). Un motivo para esta restricción es aumentar la velocidad de operación ya que las conexiones programables incurren en un retardo mayor que una conexión cableada directamente. En la Figura A.23 se muestra un ejemplo de un circuito combinacional PAL, el *AMD PAL16L8*. La letra *L* (Low) en la identificación del módulo indica que las salidas se invierten .

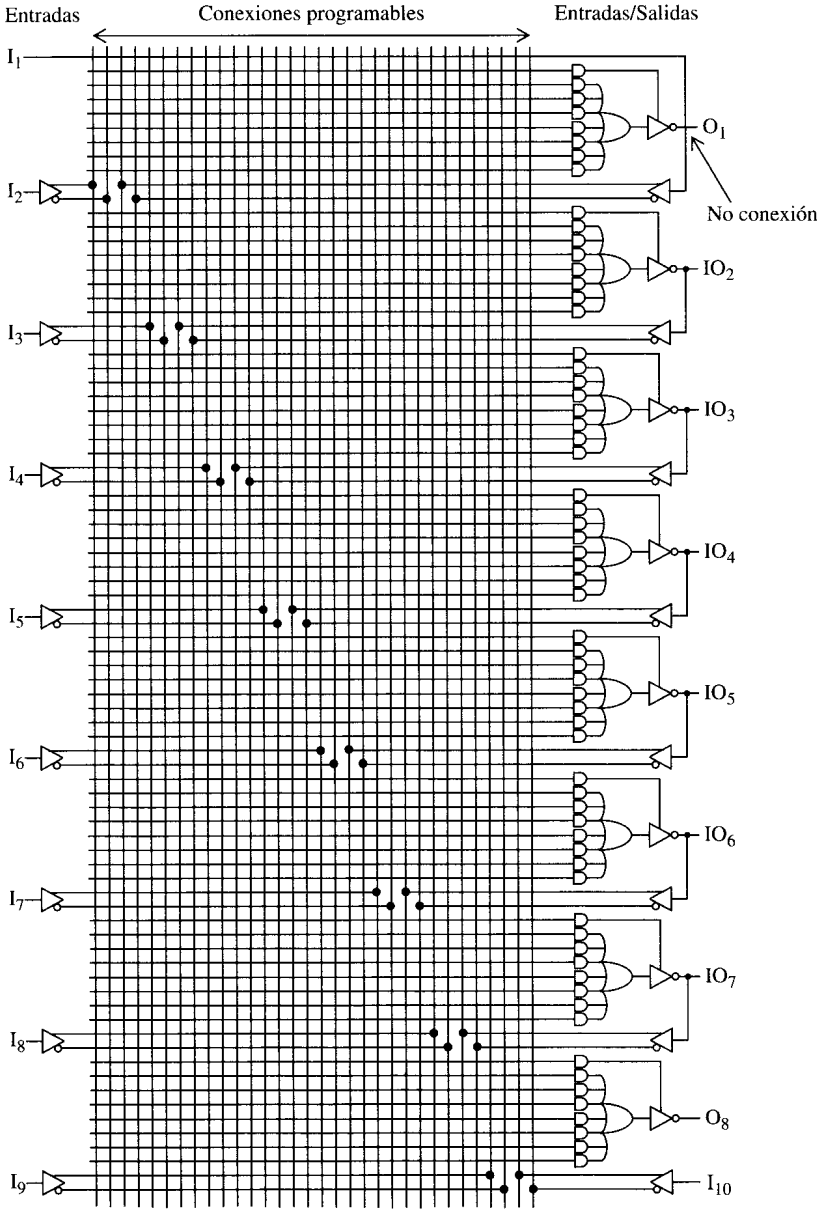


Figura A.23: Array lógico programable tipo PAL (AMD PAL16L8)

Como en el caso del PLA, inicialmente todas las conexiones están activas. Aunque en la Figura A.23 por claridad no se han dibujado. En este dispositivo hay un máximo de 16 entradas y 8 salidas, aunque algunas salidas se comparten con las entradas debido a limitaciones en el número de terminales del circuito integrado. Las conexiones de entrada/salida permiten establecer caminos de realimentación. Las salidas del módulo van dotadas de puertas triestado que se pueden activar mediante un término producto de las entradas y variables de realimentación. Las conexiones de entrada/salida compartidas se pueden emplear como entradas cuando se desactivan las salidas.

En la Figura A.23, hay 10 entradas independientes, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9 e I_{10}, cada una de ellas con posibilidad de selección de la variable o de su complemento. Hay 8 salidas, 6 de las cuales también sirven como entradas IO_2, IO_3, IO_4, IO_5, IO_6, IO_7 y 2 se dedican a salidas O_1 y O_8. Este módulo puede generar 8 sumas de funciones producto, cada una con 7 términos producto y cada término producto con 10 variables. Alternativamente se pueden generar 2 sumas de términos productos cada una con 7 términos productos y hasta 16 variables (6 de las cuales provienen de los terminales compartidos de entrada/salida). Son posibles otras configuraciones aunque siempre sujetas a las limitaciones que imponen la compartición de líneas de entrada/salida. En este caso no hay ninguna ventaja en tener términos productos compartidos entre las expresiones de suma de productos ya que cada término se genera separadamente para cada puerta OR. Como en el *PAL16L8* se invierten las salidas, las funciones lógicas que implementan son en realidad del tipo AND-OR-NOT.

A.6.4 Ejemplo: Síntesis de funciones lógicas con un PAL

La Figura A.24 muestra como se implementaría la función $\bar{f} = \bar{a}bc + a\bar{b}c + \bar{a}\bar{c}$ en una salida de un *PAL16L8*. Se observa que las puertas AND que no se utilizan tienen intactas todas sus conexiones lo que fuerza un 0 a las entradas de estas puertas. Las salidas de las puertas AND estarán así en un 0 permanente y no afectan a la puerta OR correspondiente. La salida $\bar{f}$ se genera eliminando todas las conexiones a la entrada de la puerta AND que controla la puerta triestado de salida ya que se precisa un 1 para activar dicha puerta.

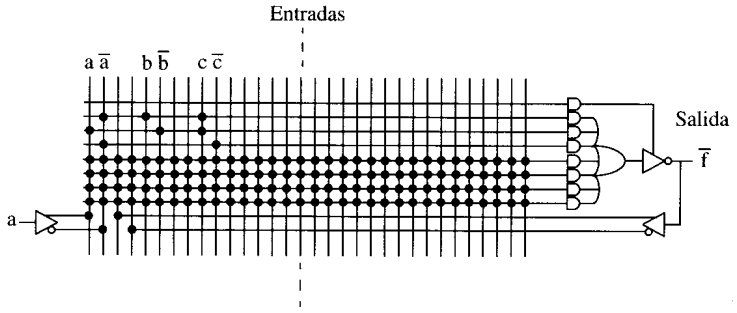


Figura A.24: Implementación de la función $\bar{f} = \bar{a}bc + a\bar{b}c + \bar{a}\bar{c}$ utilizando un PAL

Para permitir que la síntesis de una función lógica se pueda realizar en un PAL específico, una expresión suma de productos se puede implementar empleando más de dos niveles. Para hacerlo, tal como se muestra en la Figura A.25, se conecta una salida a una entrada.

Por ejemplo para obtener $a + b + c + d + e + f + g + h + i + j$, la primera etapa podría generar $x = a + b + c + d + e + f + g$ y la segunda etapa $x + h + i + j$. En la Figura A.25 no se incluyen, para una mayor claridad, las conexiones a las puertas AND no utilizadas.

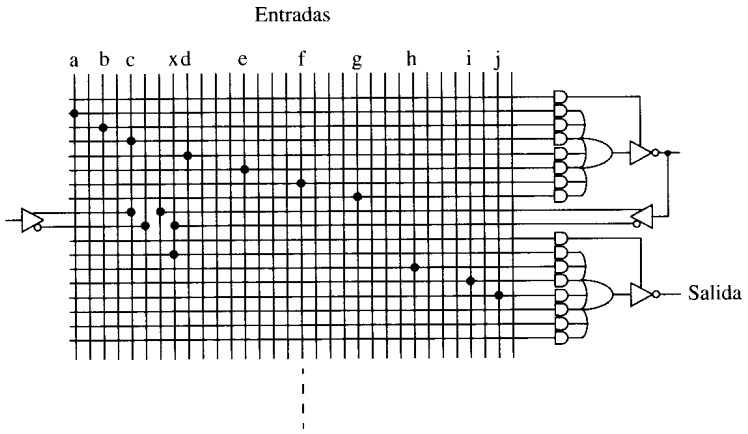


Figura A.25: Expresión multinivel de una suma de productos

El *AMD PAL16L8* descrito en la Figura A.23 proporciona la salida complementada (la *L* en el nombre indica que la salida es la complementaria). Existen PAL´s con salida sin complementar (emplean en el nombre la letra *H*). Si se dispone de un PAL con salida complementada y lo que se desea es implementar la función sin complementar la forma más sencilla de deducir la inversa de la función como una suma de términos productos es agrupar los 0´s de la función en un diagrama de Karnaugh en lugar de los 1´s.

En algunos dispositivos es posible programarlos de forma que pueden generar tanto la función sin complementar como complementada. Tal como se muestra en la Figura A.26, estos dispositivos incorporan un circuito programable que permite realizar ambas posibilidades.

Así, si se deja intacta la conexión programable se obtiene que *salida* = *entrada*, mientras que si se elimina dicha conexión se genera *salida* = $\overline{entrada}$. El circuito de la Figura A.26 se colocaría después de la puerta OR y antes del buffer triestado.

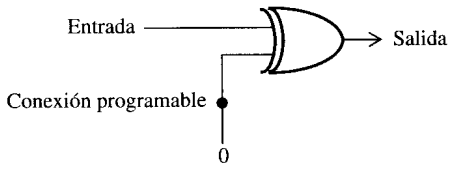


Figura A.26: Circuito de salida verdadera/inversa

Apéndice B

Introducción a los circuitos secuenciales

Los circuitos combinacionales no nos permiten por si mismos resolver el problema de almacenar el valor que tienen las entradas en un instante y utilizarlas posteriormente para tomar decisiones cuando modifiquen sus valores.

En un circuito combinacional el valor de sus salidas depende únicamente del valor que hay presente en sus entradas. Por el contrario *en un circuito secuencial las salidas en un instante cualquiera t depende no sólo del valor actual de sus entradas sino también de sus valores previos.* Así pues, un circuito secuencial debe recordar cierta información acerca de los valores pasados de sus entradas. Esta propiedad da una gran generalidad a este tipo de circuitos y permite generar respuestas mucho más interesantes y complejas que las que produce un circuito combinacional.

La unidad de control de un computador es un ejemplo de un circuito secuencial sofisticado, pero también son circuitos secuenciales dispositivos más sencillos como una máquina de expedición automática de billetes de tren. En una máquina de este tipo se introducen secuencialmente los billetes o monedas y el resultado de la operación no depende sólo de lo último que se ha insertado sino también de lo previo. Con este amplio abanico de sistemas es necesario disponer de diferentes métodos de análisis y diseño.

Los circuitos secuenciales tienen dos usos principales en los computadores:

a) Como *elementos de memoria.* Almacenan la información mientras está siendo procesada.

b) Como *circuitos de control.* Generan las señales de control necesarias para seleccionar y habilitar una secuencia de pasos en la transferencia o en el procesamiento de datos.

Ambos circuitos, combinacionales y secuenciales, operando conjuntamente proporcionan todas las funciones de almacenamiento, transferencia, procesamiento y control necesarias en un sistema digital como es el computador.

En este apéndice, en primer lugar se formaliza la noción de máquina secuencial y se introducen las ideas básicas que permiten modelar comportamientos con esta naturaleza. El concepto fundamental es el de *estado* de una máquina secuencial, que *es toda la información que se necesita en cualquier instante de tiempo para poder conocer, dada la entrada en ese momento, cual será el valor de la salida.*

A partir de la idea de estado se presenta la estructura canónica de una máquina secuencial y se formulan los modelos de Huffman-Mealy y de Moore. También se describen los diagramas de estados y las tablas de transición de estados como formas de especificar una máquina secuencial. Se introduce el concepto de estados equivalentes y el método de las particiones para minimizar el número de estados de un circuito secuencial. Finalmente se presenta el *problema de la asignación de estados* cuyo objetivo es determinar la codificación de los estados que garantiza que el número de puertas lógicas de los circuitos combinacionales que realizan las funciones de transición f y de salida g del circuito secuencial es aceptable.

A continuación se analiza el elemento de memoria básico de los circuitos secuenciales: *el biestable*. Un biestable es un circuito secuencial con dos estados estables. Estable en este contexto quiere decir que el elemento sólo cambia de estado bajo la acción de las entradas.

Hay muy diversos tipos de elementos de memoria que difieren entre si por la función de transición de estados que realizan y a su estudio se dedica una sección. Se presentan los siguientes tipos de elementos de memoria:

1) Elemento de memoria *RS*
2) Elemento de memoria *JK*
3) Elemento de memoria *D*
4) Elemento de memoria *T*

Si se utilizan las salidas de los elementos de memoria para generar sus propias señales de entrada pueden aparecer problemas si no se emplean realizaciones robustas de los mismos. Por este motivo se introduce el elemento de memoria maestro-esclavo, que es una de las posibilidades más utilizadas en la práctica. En este tipo de elementos, las señales de entrada se muestrean durante el flanco de subida de la señal de reloj y sus salidas cambian con el flanco de bajada. Este modo de operación asegura que el elemento de memoria esclavo cambia de valor a lo más una vez durante cada período de reloj.

Las dos secciones siguientes se dedican a presentar de una manera sistemática los procedimientos de análisis y síntesis de circuitos secuenciales síncronos. El problema de análisis consiste en determinar la descripción de un sistema secuencial del que se conoce su circuito. En el problema de diseño el objetivo es obtener un circuito secuencial a partir de las especificaciones de una conducta secuencial deseada. Es por lo tanto el problema inverso del de análisis. Se incluyen numerosos ejemplos de síntesis de circuitos secuenciales modelados tanto con el modelo de Huffman-Mealy como con el modelo de Moore. Todos los diseños que se consideran pertenecen a la clase de *circuitos secuenciales síncronos* que en contraposición a los *circuitos secuenciales asíncronos* son todos aquellos que su funcionamiento está controlado por una señal de reloj externa. Los circuitos secuenciales síncronos constituyen la gran mayoría de los diseños debido a la simplicidad de las operaciones de temporización que impone la señal de reloj.

En la última sección se analiza como los dispositivos lógicos programables (PLA´s y PAL´s) que se introdujeron en el Apéndice A se pueden utilizar para crear circuitos secuenciales. En teoría las conexiones de realimentación que posee el PLA o el PAL se pueden emplear para producir elementos de memoria. Sin embargo es preferible incluir en el propio módulo elementos de memoria separados. A este circuito se le conoce como *dispositivo lógico programable con registros.*

En conjunto, el objetivo del apéndice es proporcionar una presentación lo más completa posible, aunque de caracter introductorio a los circuitos secuenciales, que proporcione los conocimientos necesarios para la síntesis de los circuitos secuenciales que posee un computador.

Sección B-1

Concepto de máquina secuencial

En un circuito combinacional no se toma en consideración la variable tiempo, ya que el estado lógico de la salida de estos circuitos sólo depende del valor de las entradas en el instante considerado y es independiente de los valores que hayan tenido dichas entradas anteriormente.

Un circuito lógico cuyas señales de salida, en un instante cualquiera, están determinadas además de por el valor presente de sus entradas por la secuencia de entradas previa, se denomina *circuito secuencial*. Es decir, la salida en el instante de tiempo t, representada por $z(t)$, depende del valor que ha tenido la entrada x en el intervalo $(-\infty, t)$. Esto se expresa como:

$$z(t) = F[x(-\infty, t)]$$

Un circuito secuencial debe tener la capacidad de capturar la influencia de las entradas pasadas sobre las salidas presentes. De la definición anterior, se hace patente la necesidad de "memorizar" la historia completa de la señal de entrada $x(-\infty, t)$ para determinar la salida en el instante t. En contraposición con los circuitos combinacionales, *la diferencia distintiva de los circuitos secuenciales es que tienen memoria*, es decir, "recuerdan" alguna característica de la secuencia de señales de entrada que han recibido.

La influencia de las señales de entrada recibidas se almacena en la memoria del circuito secuencial. De esta forma, la información contenida en la memoria constituye el *estado interno del circuito secuencial que resume todo el pasado dinámico de las entradas que ha recibido.* Dos secuencias de entrada que produzcan el mismo estado interno en el circuito secuencial son equivalentes (pertenecen a la misma clase).

Para concretar las ideas, supóngase que se dispone de un interruptor de la luz que se ha estado utilizando durante un año y se ha conmutado varias veces al día. El intentar registrar la historia de las señales de entrada recibida sería una tarea muy laboriosa. Ciertamente, el interruptor no es capaz de registrar su historia completa ni está preparado para responder a las posibles secuencias de una manera sofisticada. El conmutador lo que hace es que observa todas las secuencias de entrada de una manera muy simple: todas las secuencias que finalizan con el interruptor en una posición determinada se tratan como equivalentes (luz apagada o luz encendida). De esta manera, con cada uno de los dos conjuntos posibles de historias equivalentes se asocia un estado.

Una caja de caudales es un ejemplo de un conmutador mecánico más complejo. Su respuesta depende de una secuencia determinada de símbolos de entrada, de forma que se puede pensar en ella como si fuese una máquina secuencial. Cuando el dial de la caja de caudales recibe los tres dígitos de la combinación de forma consecutiva la puerta se abre.

Esta consideración permite ver la caja de caudales como un circuito secuencial mecánico de 4 estados y se pueden visualizar las posibles transiciones entre estados como indica la Figura B.1.

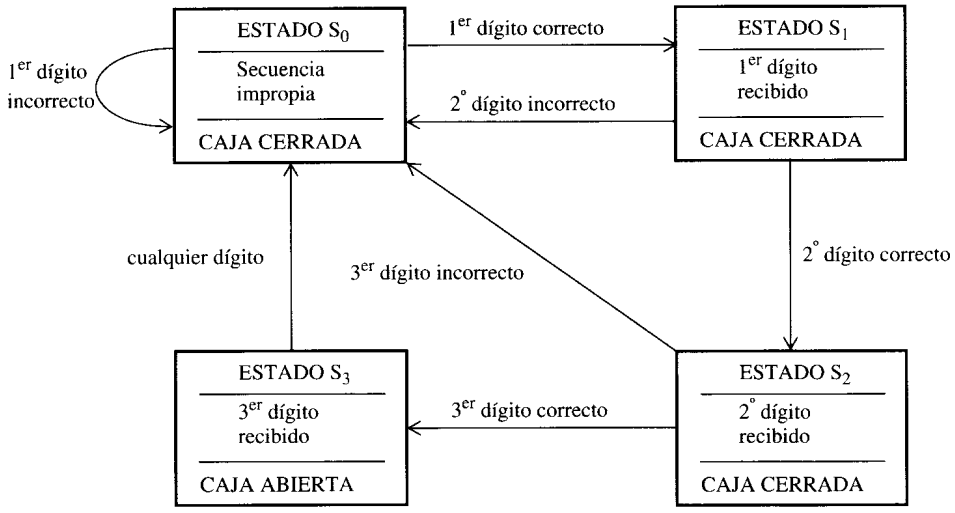


Figura B.1: Transiciones entre los estados de una caja de caudales

El estado, especificado por una variable auxiliar s, representa la clase de secuencias de entrada que son equivalentes. Resume el efecto de las entradas pasadas sobre las salidas presentes y futuras. En consecuencia, la salida z en el instante t dependerá de la entrada x y del estado s en ese mismo instante. Es decir, la *función de salida* es:

$$z(t) = g[s(t), x(t)]$$

Para que el estado continúe reflejando la influencia de las señales de entrada, su valor tiene que modificarse a lo largo del tiempo. El nuevo estado depende del estado presente $s(t)$ y de la entrada actual $x(t)$. Este cambio del estado se describe mediante la *función de transición de estado*:

$$s(t + \Delta t) = f[s(t), x(t)]$$

Los estados $s(t)$ y $s(t + \Delta t)$ se llaman *estado presente* y *próximo estado* respectivamente. Como el conjunto de estados es finito, a estos sistemas se les denomina también *máquinas de estados finitos.*

Más formalmente la descripción de estado de una máquina secuencial M viene dada por la siguiente quintupla:

$$M = (X, Z, S, f, g)$$

donde

X	*es el conjunto de símbolos de entrada*
Z	*es el conjunto de símbolos de salida*
S	*es el conjunto de estados*
$f: S \times X \rightarrow S$	*es la función de transición de estado*
$g: S \times X \rightarrow Z$	*es la función de salida*

Este modelo de máquina secuencial se conoce como *modelo de Huffman-Mealy*; su característica fundamental es que la salida en el tiempo t es función del estado y de la entrada en el mismo instante. Por otra parte en el *modelo de Moore* la salida en el tiempo t depende sólo del estado en ese instante de tiempo. La función de salida del modelo de Moore se puede expresar por:

$$z(t) = g[s(t)]$$

esto no significa que la salida es independiente de la secuencia de entrada, sino que la influencia de la entrada sobre la salida es sólo a través del estado. El ejemplo de la caja de caudales se modeló como una máquina tipo Moore.

B.1.1 Sistemas síncronos y asíncronos

Un sistema secuencial opera aceptando una entrada y produciendo un cambio en el estado de acuerdo con la función f y una salida de acuerdo con la función g.

En un sistema *secuencial síncrono* los cambios de estado tienen lugar en instantes discretos de tiempo que están definidos por una entrada de sincronización que se denomina *reloj*. En la Figura B.2 se muestra, para un sistema síncrono, la relación que existe entre la entrada, el estado y la salida como función del tiempo.

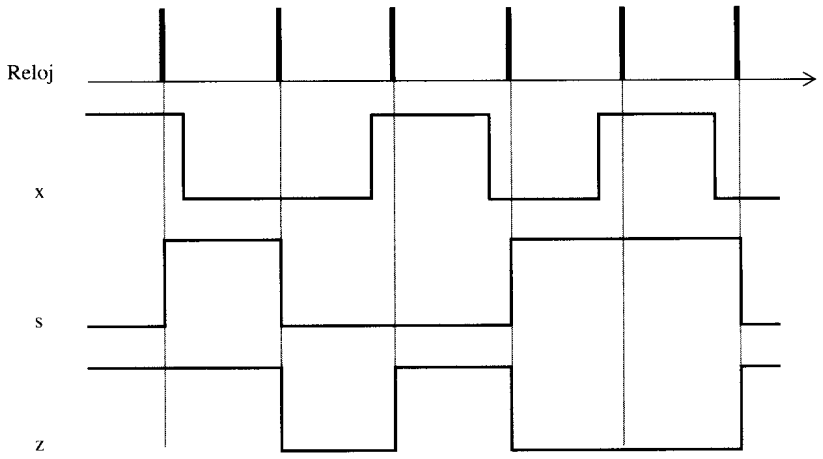


Figura B.2: Conducta síncrona de un sistema secuencial

En un sistema secuencial asíncrono el estado puede cambiar en cualquier instante en función de los cambios que se produzcan en la entrada (ver Figura B.3). Los sistemas asíncronos presentan ciertas ventajas: son más generales y normalmente más rápidos que los sistemas síncronos, ya que su funcionamiento no depende de la frecuencia de un reloj externo que en la mayoría de los casos es menor que la máxima velocidad de operación a la que pueden funcionar las puertas lógicas. La ejecución ordenada de una serie de acciones en un sistema asíncrono se controla mediante un conjunto de señales de inicialización y finalización, de manera que cuando se produce una señal de terminación comienza la ejecución de la siguiente operación en secuencia.

Sin embargo, la mayor parte de los circuitos secuenciales que se utilizan en el diseño de sistemas digitales son de tipo síncrono, debido a que son más fáciles de diseñar y son más fiables que los asíncronos. Por esta razón, en lo que sigue fundamentalmente se considera el modo de operación síncrona. La señal de reloj permite sincronizar todas las transiciones de estado, de forma que el intervalo entre dos cambios sucesivos es igual a un período de reloj. Al utilizar de esta forma las señales de reloj se simplifica en gran medida el diseño y el funcionamiento de los circuitos secuenciales, pues se pueden ignorar pequeñas diferencias en los retardos y en la propagación de señales en las diferentes partes del circuito. Esto provoca que la frecuencia del reloj, sea lo suficientemente baja como para que todos los elementos del circuito tengan tiempo de realizar el cambio al nuevo estado. De esta forma, un sistema síncrono funciona como máximo a la velocidad del elemento más lento. Por este motivo son más lentos que los sistemas asíncronos, en los que se aprovecha mejor las velocidades altas de algunos elementos ya que no tienen que esperar a la señal de reloj.

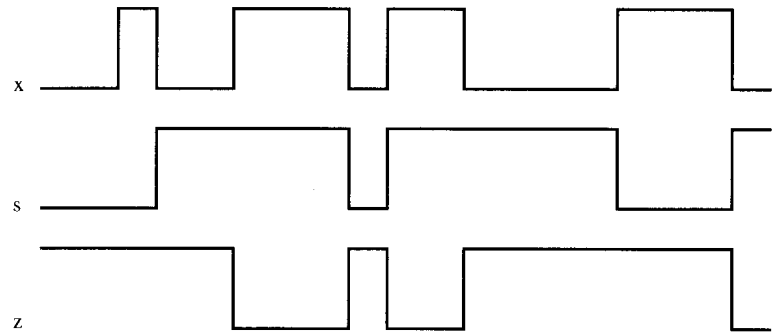


Figura B.3: Conducta asíncrona de un sistema secuencial

Como en los sistemas secuenciales síncronos los cambios en el estado tienen lugar en instantes discretos definidos, es posible sustituir la variable continua t por una variable discreta *i*, (*i* = 1, 2 ....). Las funciones que describen un sistema asíncrono se pueden expresar como:

$$z_i = g[s_i, x_i] \qquad \text{Función de salida}$$

$$s_{i+1} = f[s_i, x_i] \qquad \text{Función de transición}$$

B.1.2 Estructura canónica de una máquina secuencial

La estructura canónica de una máquina secuencial se basa en su descripción de estado. Consta de un registro de estado que almacena el valor del estado y un circuito combinacional que realiza las funciones de transición *f* y de salida *g*.

Esta estructura canónica general adopta una forma algo diferente según se trate de una máquina secuencial tipo Huffman-Mealy o Moore. En la Figura B.4 se muestra la diferencia, entre ambos tipos de máquinas que afecta a la generación de la función de salida. En el primer caso depende tanto de la entrada como del estado presente y en el segundo sólo es función del estado presente. La señal de reloj determina los instantes de tiempo en los que el nuevo estado se carga en el registro. La señal de inicializar permite poner el registro de estado en su valor inicial.

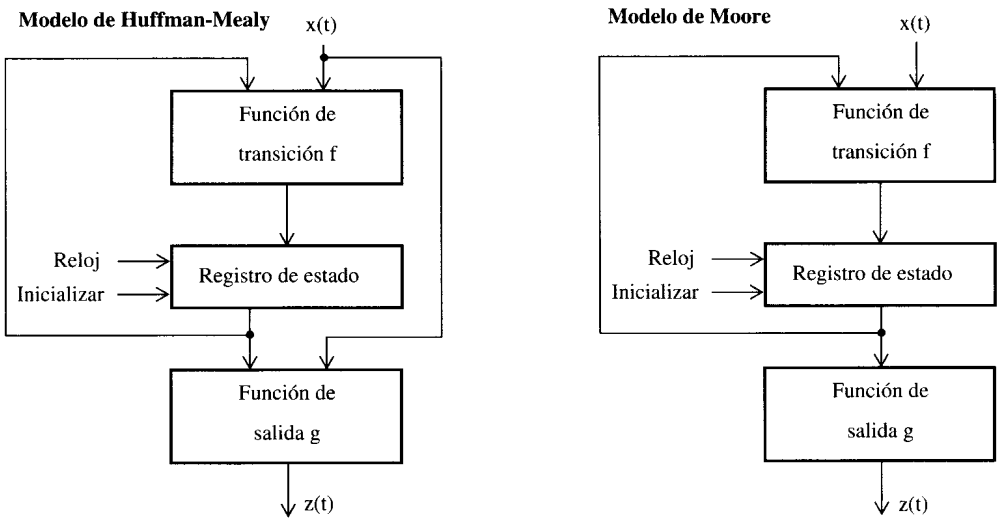


Figura B.4: Realización canónica de máquinas secuenciales tipo Huffman-Mealy y Moore

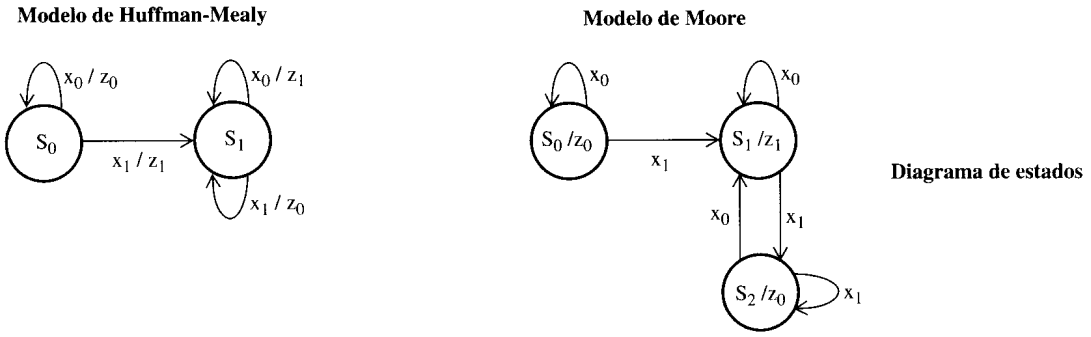
B.1.3 Diagrama de estados y tabla de estados

Las funciones de transición de estado f y de salida g de una máquina secuencial se suelen representar mediante un diagrama de estados o una tabla de estados. Un *diagrama de estados* es un *grafo dirigido* que representa gráficamente el comportamiento de un circuito secuencial. Cada *nodo* se corresponde con un estado del sistema y cada *arco* entre dos nodos representa una transición entre estados producida por alguna entrada del sistema. Por tanto, cada nodo y cada arco estarán etiquetados con un estado y una entrada respectivamente. Las salidas se representarán en los nodos o en los arcos, dependiendo de que se trate de una máquina secuencial tipo Moore o Huffman-Mealy (ver Figura B.5).

Figura B.5: Representación mediante diagrama de estados

La *tabla de estados* se obtiene automáticamente a partir del diagrama de estados. En las filas y columnas se disponen los estados y entradas presentes. En una máquina secuencial tipo Huffman-Mealy en la

intersección de cada fila y columna se coloca el próximo estado y la salida que corresponde. Las salidas en el instante t dependen de las entradas en ese mismo instante. En una máquina secuencial tipo Moore en la intersección de cada fila y columna se dispone sólo el próximo estado y asociado con cada estado hay una salida. Las salidas en el instante t dependen de las entradas en el instante t-1 (a través del estado en el instante t). En la Figura B.6 se muestra un ejemplo de ambos tipos de diagramas de estados y tablas de estados.



Estado presente	Entrada	
	x_0	x_1
S_0	S_0 / z_0	S_1 / z_1
S_1	S_1 / z_1	S_1 / z_0
	Próximo estado / Salida	

Estado presente	Entrada		Salida
	x_0	x_1	
S_0	S_0	S_1	z_0
S_1	S_1	S_1	z_1
S_2	S_1	S_2	z_0
	Próximo estado		

Tabla de estados

Figura B.6: Ejemplo de diagrama y tabla de estados de un modelo de Huffman-Mealy y de Moore

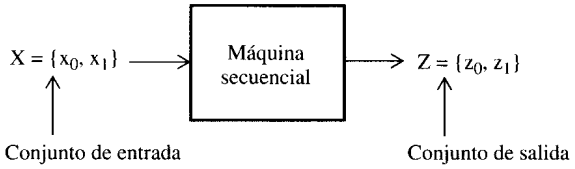
B.1.4 Minimización del número de estados

El objetivo de una máquina secuencial es proporcionar desde el punto de vista de sus entradas y salidas una conducta temporal determinada. Bajo esta consideración, el problema de minimización del número de estados se puede formular en los términos siguientes:

Dada una máquina secuencial M_1 (definida por su tabla de estados), ¿es posible obtener otra máquina secuencial M_2 que tenga un número de estados menor que M_1 y que posea idéntica conducta de entrada-salida?. Esta pregunta tiene importancia puesto que afecta directamente a la cantidad de memoria que será necesaria utilizar en la síntesis de una máquina secuencial.

El procedimiento de minimización del número de estados que se expone a continuación se debe a Huffman y permite obtener de forma sistemática el diagrama de estados mínimo a partir de uno inicial dado. Se fundamenta en el concepto de *estados equivalentes* y en las condiciones necesarias y suficientes para que dos estados se puedan considerar equivalentes. De forma general se dice que dos estados son equivalentes si no es posible diferenciarlos. En otras palabras, no se puede determinar, aplicando secuencias de entrada y observando sus salidas, desde cual de los dos estados equivalentes ha comenzado a operar la máquina secuencial. Si esta condición se verifica para todas las secuencias de entrada, uno de los dos estados es redundante y puede eliminarse sin alterar el comportamiento de la máquina secuencial. El ejemplo que se

presenta a continuación sirve para introducir la idea de equivalencia. En la Figura B.7 se muestra la tabla de estados de una máquina secuencial M de una entrada y una salida .



Estado presente	Entrada x_0	Entrada x_1
S_0	S_2 / z_1	S_1 / z_0
S_1	S_2 / z_1	S_4 / z_0
S_2	S_1 / z_1	S_4 / z_0
S_3	S_3 / z_0	S_1 / z_1
S_4	S_4 / z_0	S_0 / z_1
	Próximo estado / Salida	

Figura B.7: Diagrama de estados de una máquina secuencial

Se supone que se desconoce el estado inicial en el que se encuentra la máquina. Si se aplica una entrada $x = x_0$ y se obtiene como salida $z = z_1$, se puede afirmar que el estado inicial era S_0, S_1 ó S_2. Si la salida es $z = z_0$ el estado inicial puede ser S_3 o S_4. A una condición análoga se llega si la entrada es $x = x_1$. Se concluye pues que los estados S_0, S_1 y S_2 de una parte y S_3 y S_4 de otra son equivalentes para secuencias de entrada de longitud uno, es decir son *1-equivalentes*. En la Figura B.8 se muestra la conducta de la máquina para secuencias de entrada de longitud 2 y 3. Se observa que los estados (S_1, S_2) y (S_3, S_4) son *2-equivalentes* y que (S_3, S_4) son también *3-equivalentes*. En general se puede demostrar que (S_3, S_4) son *k-equivalentes* $\forall$ *k*.

Estado inicial	Secuencia de entradas $x_0\,x_0$	$x_0\,x_1$	$x_1\,x_0$	$x_1\,x_1$
S_0	$z_1\,z_1$	$z_1\,z_0$	$z_0\,z_1$	$z_0\,z_0$
S_1	$z_1\,z_1$	$z_1\,z_0$	$z_0\,z_0$	$z_0\,z_1$
S_2	$z_1\,z_1$	$z_1\,z_0$	$z_0\,z_0$	$z_0\,z_1$
S_3	$z_0\,z_0$	$z_0\,z_1$	$z_1\,z_1$	$z_1\,z_0$
S_4	$z_0\,z_0$	$z_0\,z_1$	$z_1\,z_1$	$z_1\,z_0$
	Secuencia de salidas			

Estado inicial	Secuencia de entradas $x_0\,x_0\,x_0$	$x_0\,x_0\,x_1$	$x_0\,x_1\,x_0$	$x_0\,x_1\,x_1$	$x_1\,x_0\,x_0$	$x_1\,x_0\,x_1$	$x_1\,x_1\,x_0$	$x_1\,x_1\,x_1$
S_0	$z_1\,z_1\,z_1$	$z_1\,z_1\,z_0$	$z_1\,z_0\,z_0$	$z_1\,z_0\,z_1$	$z_0\,z_1\,z_1$	$z_0\,z_1\,z_0$	$z_0\,z_0\,z_0$	$z_0\,z_0\,z_1$
S_1	$z_1\,z_1\,z_1$	$z_1\,z_1\,z_0$	$z_1\,z_0\,z_0$	$z_1\,z_0\,z_1$	$z_0\,z_0\,z_0$	$z_0\,z_0\,z_1$	$z_0\,z_1\,z_1$	$z_0\,z_1\,z_0$
S_2	$z_1\,z_1\,z_1$	$z_1\,z_1\,z_0$	$z_1\,z_0\,z_0$	$z_1\,z_0\,z_1$	$z_0\,z_0\,z_0$	$z_0\,z_0\,z_1$	$z_0\,z_1\,z_1$	$z_0\,z_1\,z_0$
S_3	$z_0\,z_0\,z_0$	$z_0\,z_0\,z_1$	$z_0\,z_1\,z_1$	$z_0\,z_1\,z_0$	$z_1\,z_1\,z_1$	$z_1\,z_1\,z_0$	$z_1\,z_0\,z_0$	$z_1\,z_0\,z_1$
S_4	$z_0\,z_0\,z_0$	$z_0\,z_0\,z_1$	$z_0\,z_1\,z_1$	$z_0\,z_1\,z_0$	$z_1\,z_1\,z_1$	$z_1\,z_1\,z_0$	$z_1\,z_0\,z_1$	$z_1\,z_0\,z_0$
	Secuencia de salidas							

Figura B.8: Estados 2-equivalentes y 3-equivalentes de la máquina secuencial de la Figura B.7

El ejemplo anterior permite formalizar la noción de estados equivalentes:

Definición I

Los estados S_1, S_2, ..., S_j de una máquina secuencial son equivalentes si y sólo si para toda posible secuencia de entrada se produce la misma secuencia de salida, independientemente de que el estado inicial sea S_1, S_2, ..., S_j. Esta definición se puede expresar de una forma alternativa para parejas de estados.

Definición II

Sean S_k y S_l los próximos estados de una máquina secuencial cuando se le aplica la entrada x_p a la máquina en los estados S_i y S_j respectivamente (ver Figura B.9). Los estados S_i y S_j son equivalentes si y sólo si $\forall$ posible entrada x_p se verifica:

1) La salida producida por el estado S_i es igual a la salida producida por S_j.

2) Los próximos estados S_k y S_l son equivalentes.

Figura B.9: Transiciones de estado desde S_i y S_j bajo una misma entrada

La Definición II se puede deducir de la Definición I de la forma siguiente: Si S_i produce una salida diferente de S_j para alguna entrada x_p, S_i y S_j no pueden ser equivalentes (no serían 1-equivalentes) y la primera condición es necesaria.

Si S_k y S_l no son estados equivalentes, existirá una secuencia de entrada x_1, x_2, ..., x_m que produce secuencias de salida diferentes con S_k y S_l como estados iniciales. Por lo tanto x_p, x_1, x_2, ..., x_m dará también una secuencia de salida diferente para S_i y S_j como estados iniciales. Es decir, S_i y S_j no pueden ser equivalentes si no se verifica la segunda condición. Las dos condiciones de la Definición II constituyen la base de la reducción de estados por el *método de las particiones*. Hay que determinar sucesivas particiones P_k, $k = 1, 2, ...$ definidas de la forma siguente:

$$P_k = \{\text{Bloques}\} \qquad \text{Bloque} = \{\text{Estados k-equivalentes}\}$$

El proceso de particionamiento se efectua en dos pasos:

Paso 1 La primera partición P_1 se forma colocando en el mismo bloque los estados que son 1-equivalentes (es decir, si sus salidas son idénticas para todas las entradas).

Paso 2 Las particiones P_k, $k = 2, 3, ...$ se obtienen colocando dos o más estados en el mismo bloque de P_k si y sólo si para todas las entradas posibles, sus próximos estados pertenecen a un mismo bloque de P_{k-1}. Si $P_{k+1} = P_k$ para algún $k \Rightarrow P_j = P_k \; \forall \, j > k$ (la demostración por inducción es inmediata). El proceso termina pues cuando coinciden dos particiones sucesivas.

Este esquema iterativo siempre finaliza, porque para una máquina secuencial con n estados existen como máximo n - 1 particiones diferentes. Esto se deduce fácilmente si se observa que la partición P_{k+1} es un refinamiento de P_k y por lo tanto si todas las particiones hasta P_{n-1} son diferentes, entonces P_{n-1} tendrá n bloques con un estado en cada bloque. Para el ejemplo anterior la partición P_1 es:

$$P_1 = (S_0, S_1, S_2)(S_3, S_4)$$

ya que los estados que están en cada bloque son 1-equivalentes.

Para determinar la partición P_2, se analizan los próximos estados de los bloques (S_0, S_1, S_2) y (S_3, S_4). De la tabla de estados de la Figura B.7 se obtiene (ver Figura B.10):

Bloque 1

Estado presente	Entrada x_0	Entrada x_1
S_0	S_2	S_1
S_1	S_2	S_4
S_2	S_1	S_4
	Próximo estado	

Bloque 2

Estado presente	Entrada x_0	Entrada x_1
S_3	S_3	S_1
S_4	S_4	S_0
	Próximo estado	

Figura B.10: Determinación de la partición P_2

Hay que considerar ahora las columnas de estas tablas para ver si sus estados son 1-equivalentes. En la tabla de la izquierda cuando se aplica la entrada x_1 los próximos estados que resultan (S_1 y S_4) no son 1-equivalentes, por lo que el bloque (S_0, S_1, S_2) se tiene que subdividir en dos subbloques (S_0) y (S_1, S_2). Los próximos estados del bloque (S_3, S_4) son 1-equivalentes para cada una de las entradas. En consecuencia la partición P_2 es:

$$P_2 = (S_0)(S_1, S_2)(S_3, S_4)$$

De la misma forma, comprobando los estados sucesores de (S_1, S_2) y (S_3, S_4) se obtiene P_3:

$$P_3 = (S_0)(S_1, S_2)(S_3)(S_4)$$

Análogamente se obtiene P_4 a partir de P_3:

$$P_4 = (S_0)(S_1, S_2)(S_3)(S_4)$$

como $P_4 = P_3$, esta es la partición de equivalencia y los estados S_1 y S_2 son equivalentes y uno de ellos se puede suprimir de la tabla de estados original. En la Figura B.11 se muestra la tabla de estados mínima que resulta. Se ha suprimido la fila del estado S_2 y en todas las entradas que aparece S_2 se pone S_1.

El Paso 2 del procedimiento y sucesivos se puede facilitar si a los bloques de P_k se les asigna un número y debajo de cada estado se escriben los números correspondientes de sus próximos estados. Dos estados estarán en el mismo bloque de P_{k+1} si sus columnas de próximos estados son idénticas.

En la Figura B.12 se muestra, para el ejemplo anterior, como se obtiene la partición P_2.

Estado presente	Entrada	
	x_0	x_1
S_0	S_1 / z_1	S_1 / z_0
S_1	S_1 / z_1	S_4 / z_0
S_3	S_3 / z_0	S_1 / z_1
S_4	S_4 / z_0	S_0 / z_1
	Próximo estado / Salida	

Figura B.11: Tabla de estados mínima del ejemplo de la Figura B.8

Partición	1			2	
P_1	(S_0,	S_1,	S_2)	(S_3,	S_4)
x_0	1	1	1	2	2
x_1	1	2	2	1	1

$\Rightarrow \quad P_2 = (S_0)\,(S_1, S_2)\,(S_3, S_4)$

Figura B.12: Procedimiento simplificado de obtención de la partición P_2

B.1.5 Codificación binaria y el problema de la asignación de estados

La descripción que se ha efectuado de una máquina secuencial, hasta el momento, se ha expresado dándole un significado de alto nivel a los elementos que constituían los conjuntos de entrada X, de estado S y de salida Z. Sin embargo, la realización física de un sistema digital utiliza variables binarias. Por este motivo la descripción de alto nivel se convierte a una descripción binaria codificando las entradas, salidas y estados. Las variables x, z y s se codifican como vectores binarios de la forma siguiente:

1) *Función de codificación de las entradas* C_x

$$C_x: \quad X \rightarrow \{0, 1\}^n \quad n \geq \log_2 |X|$$

donde n es el número de variables binarias de entrada y $|X|$ representa el cardinal del conjunto X (número de elementos de X).

2) *Función de codificación de las salidas* C_z

$$C_z: \quad Z \rightarrow \{0, 1\}^m \quad m \geq \log_2 |Z|$$

donde m es el número de variables binarias de salida

3) *Función de codificación del estado* C_s

$$C_s: \quad S \rightarrow \{0, 1\}^k \quad k \geq \log_2 |S|$$

donde k es el número de variables binarias de estado

Son posibles muchos esquemas de codificación para las variables de entrada, salida y estado. En general la codificación de X y Z depende de condiciones externas a la máquina secuencial, ya que las señales de entrada y salida están conectadas a otros circuitos. La codificación del estado de la máquina es lo que se conoce como *problema de la asignación de estados*. Históricamente este problema ha tenido una gran importancia y su objetivo es encontrar la codificación que garantiza que el circuito combinacional que realiza las funciones de transición f y de salida g tiene un coste mínimo. De forma general no ha sido aún resuelto, y aunque sigue permaneciendo como un reto teórico su interés desde un punto de vista práctico, ha quedado muy reducido por la aparición de los Arrays Lógicos Programables (PLA), las memorias ROM y demás circuitos integrados con un elevado nivel de integración.

No obstante existen algunas reglas heurísticas que sin garantizar una solución óptima suelen dar resultados aceptables. Se basan en el hecho conocido de que se obtienen funciones de transición f y de salida g más simples si los 1's de dichas funciones ocupan celdas adyacentes en un mapa de Karnaugh. Estas reglas por orden de prioridad son las siguientes:

Regla 1. Asignar códigos que difieren en una variable a los estados presentes que llevan al sistema al mismo próximo estado. En la Figura B.13 se aplica esta regla a la tabla de estados mínima de la Figura B.11.

Figura B.13: Aplicación de la Regla 1 al ejemplo de la Figura B.11

De la Figura B.13 se sigue que de una parte los estados (S_0, S_1) y de otra (S_0, S_3) deben tener codificaciones adyacentes. Para este ejemplo sólo con esta regla queda completamente determinada la asignación de estados. En la Figura B.14 se muestra una posible asignación de estados.

	Q_1	Q_2
$S_0 \rightarrow$	0	0
$S_1 \rightarrow$	0	1
$S_3 \rightarrow$	1	0
$S_4 \rightarrow$	1	1

Figura B.14: Asignación de estados que resulta de aplicar la Regla 1 al ejemplo de la Figura B.11

Regla 2. Asignar códigos que difieren en una variable a los próximos estados de un estado presente. Aunque en el ejemplo anterior no es ya preciso utilizar esta regla, en la Figura B.15 se muestra cual sería el resultado de su aplicación en el caso de que el estado presente sea S_1 o S_3.

Figura B.15: Aplicación de la Regla 2 al ejemplo de la Figura B.11

La aplicación de esta regla implicaría que los estados (S_1, S_4) y (S_1, S_3) deberían tener codificaciones adyacentes. Si se observa la asignación de estados de la Figura B.14 esto se verifica para la primera pareja de estados pero no así para la segunda. Sin embargo como se ha dicho la regla 1 tiene preferencia sobre la regla 2.

Regla 3. Asignar códigos que difieren en una variable a los estados presentes con las mismas salidas. La aplicación de esta regla implicaría que los estados (S_0, S_1) y (S_3, S_4) deberían tener codificaciones adyacentes. Ambas condiciones se cumplen para la asignación de estados que se muestra en la Figura B.14.

En la Figura B.16 se da la realización canónica de una máquina secuencial codificada en binario.

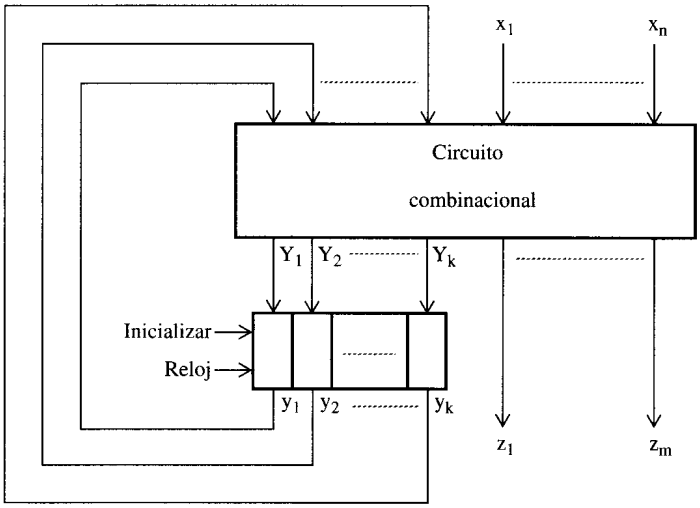


Figura B.16: Realización canónica de una máquina secuencial con variables binarias

El registro de estado está formado por k celdas binarias. El estado presente se representa por y_1, y_2, ..., y_k y el próximo estado por Y_1, Y_2, ..., Y_k.

Sección B-2

Elemento de memoria

¿Es posible interconectar puertas lógicas de manera que la salida del circuito en cualquier instante esté determinada, además de por los valores presentes en sus entradas, por los valores previos que éstas hayan tenido?. Para contestar a la pregunta se va a analizar el circuito lógico que se muestra en la Figura B.17.

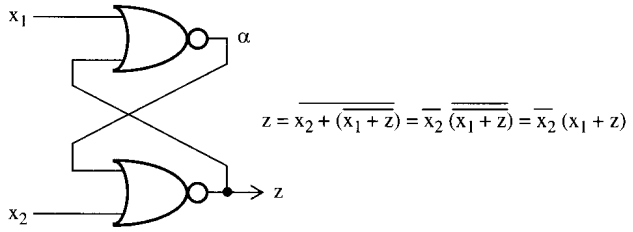


Figura B.17: Elemento de memoria básico realizado con puertas NOR

Se desea encontrar una expresión lógica que determine z como función de sus entradas x_1 y x_2. La ecuación de la Figura B.17 no da z de forma explícita (ya que aparece a ambos lados del signo igual).

Si $x_1 = x_2 = 0$, la ecuación anterior se reduce a $z = z$ y por lo tanto $z = 0$ o $z = 1$ es una solución. En este caso el valor de z no está unívocamente determinado por x_1 y x_2. El mismo resultado se puede obtener de otra forma. Si $x_1 = x_2 = 0$ el circuito de la Figura B-15 se puede reemplazar por el de la Figura B.18.

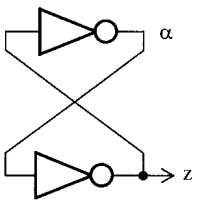


Figura B.18: Circuito equivalente al de la Figura B.17 cuando $x_1 = x_2 = 0$

Éste es un circuito *autónomo* (no posee señales de entrada) y x_1 y x_2 no determinan cual es el valor de z, ya que no aparecen. ¿Qué valor tendrá z?. Una forma de contestar a esta cuestión es admitir una solución y verificarla. Si se supone $z = 0$, se encuentra que $\alpha = 1$ y $z = 0$, por lo que se puede afirmar que el circuito de la Figura B.18 preserva el valor de z. Análogamente, si se postula $z = 1$, se obtiene $\alpha = 0$ y $z = 1$ y también se

mantiene este valor. Por supuesto z no puede tener ambos valores simultáneamente ¿Qué es entonces lo que determina el valor de z en un instante específico? Para ser capaces de determinar el valor que posee z, en la Figura B.19 se propone un modelo más realista del circuito de la Figura B.17. Después de cada puerta lógica se ha introducido un elemento de retardo ideal τ, que se supone que reproduce fidedignamente en su terminal de salida la señal que estaba presente en la entrada τ segundos antes. La señal de entrada está restringida a efectuar como máximo un cambio durante cualquier intervalo de τ segundos. La Figura B.19 corresponde a un *modelo detallado* del circuito mientras que la Figura B.17 representa un *modelo ideal o instantáneo.*

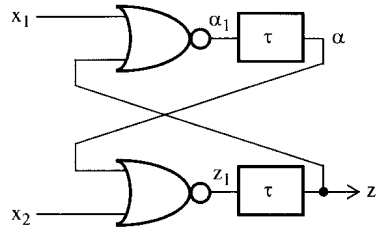


Figura B.19: Modelo detallado del elemento de memoria básico

Para el circuito de la Figura B.19 se supone que $x_1 = x_2 = z = 0$ y que estas condiciones han persistido por largo tiempo de forma que $\alpha = 1$. En estas circunstancias las señales de entrada y de salida de los elementos de retardo tienen los mismos valores, es decir, $z_1 = z$ y $\alpha_1 = \alpha$ y el circuito está en un *estado estable* ya que ninguna señal tiende a cambiar su valor. A partir de este estado estable si se hace $x_1 = 1 \Rightarrow \alpha_1 = 0$ y τ segundos después $\alpha = 0$. Durante este intervalo $\alpha_1 \neq \alpha$ (aunque finalmente α cambia su valor) de manera que el circuito está en un *estado inestable*. Cuando $\alpha = 0 \Rightarrow z_1 = 1$ lo que provoca que el circuito esté otra vez en un estado inestable. Transcurridos τ segundos más $z = 1$, lo que no altera ya el valor de α_1 y el circuito ha alcanzado otro de sus estados estables, en el que permanecerá mientras las entradas sean $x_1 = 1$ y $x_2 = 0$.

Si se retorna ahora x_1 al valor 0 ($x_1 = x_2 = 0$) ni z ni α cambian su valor. El razonamiento expuesto muestra que al hacer $x_1 = 1$ durante un corto intervalo de tiempo, se origina que el circuito conmute de un estado estable con $z = 0$ a otro estado estable con $z = 1$.

Un estudio similar demuestra que haciendo $x_2 = 1$ por un tiempo mayor que 2τ segundos, provoca que el circuito vuelva a conmutar al estado estable $\alpha = 1, z = 0$ y que permanezca en ese valor cuando x_2 vuelve a 0. El análisis anterior se puede resumir de la forma siguiente:

$$(x_1, x_2) = (1, 0) \quad \Rightarrow \quad (\alpha, z) = (0, 1)$$

$$(x_1, x_2) = (0, 1) \quad \Rightarrow \quad (\alpha, z) = (1, 0)$$

Se ve pues que, *lo que determina el valor de z cuando* $x_1 = x_2 = 0$, *es cuál de las dos entradas* (x_1 o x_2) *ha sido la última en estar a* 1. Se tiene así una versión electrónica del conmutador de la luz. El circuito de la Figura B.17 es un circuito secuencial que posee 2 estados ("recuerda" que variable de entrada tiene o ha tenido más recientemente el valor 1) y es un tipo particular de *elemento de memoria* (se conoce también como biestable o flip-flop).

Existe sin embargo una situación excepcional de este elemento de memoria que debe analizarse con cuidado y que corresponde a que ambas entradas valen 1, es decir:

$$(x_1, x_2) = (1, 1) \quad \Rightarrow \quad (\alpha, z) = (0, 0)$$

el circuito está en un estado estable singular que es poco deseable. Se evita este estado imponiendo la condición de que x_1 y x_2 no pueden tener simultáneamente el valor 1 (x_1 AND $x_2 = 0$).

¿Pero por qué no es recomendable el estado $\alpha = z = 0$? En primer lugar, si se utiliza α como el complemento de z en un circuito combinacional que está controlado por este circuito secuencial, cuando α tiene el mismo valor que z no puede esperarse que de resultados correctos. En segundo lugar y más importante, si se supone que después de $x_1 = x_2 = 1$, x_1 y x_2 toman al mismo tiempo el valor 0 no es posible predecir el estado estable que el circuito finalmente alcanzará. Pequeñas variaciones en los parámetros físicos del circuito pueden hacer que éste vaya siempre al estado estable $z = 0$, $\alpha = 1$ o $z = 1$, $\alpha = 0$. Esto depende de cual de los dos retardos que se han asociado con las puertas lógicas es menor y por lo tanto inicia el proceso de transición entre estados. De esta forma, circuitos aparentemente idénticos pueden no ir al mismo estado estable. Es precisamente esta conducta no determinista la que debe evitarse. Es necesario conocer exactamente lo que los circuitos hacen y que circuitos del mismo tipo funcionan igual cada vez que se encuentran en una situación idéntica.

El circuito secuencial que se ha considerado es de tipo asíncrono, ya que los cambios en la salida se producen como respuesta a cambios en las entradas (no existe una señal de sincronización). Para poner más claramente esta relación en la Figura B.20 se muestra el mismo circuito de la Figura B.19 (por generalidad se supone ahora que los retardos asociados con ambas puertas son distintos τ_1 y τ_2) de acuerdo con la estructura dada a una máquina secuencial en la Figura B.16.

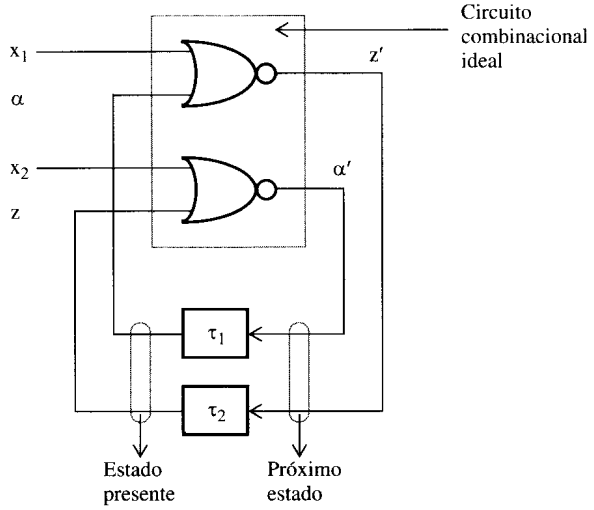


Figura B.20: Elemento de memoria dispuesto con la estructura de una máquina secuencial

En cualquier instante las entradas a los elementos con retardo quedan determinadas por los valores que tienen a sus salidas y el valor actual de las señales de entrada. El estado presente se corresponde con las señales de salida de los elementos con retardo y el próximo estado con sus entradas.

Para asegurar que el estado presente será seguido finalmente por el próximo estado, se ha visto que es necesario suponer que ninguna señal de entrada cambia su valor dos veces durante un intervalo de τ segundos

($\tau = \max(\tau_1, \tau_2)$). Si no se intentan igualar estos retardos, o no se fuerzan los instantes de tiempo en los que cambian las entradas y el estado del circuito, se dice que está funcionando de manera asíncrona.

Cuando se impone que todos los retardos del circuito son iguales y que las señales de entrada y el estado modifican su valor al mismo tiempo, se está operando en modo síncrono. Esto se puede conseguir si se sincronizan todos los cambios de estado con una señal de reloj. En la Figura B.21 se muestra como el circuito asíncrono de la Figura B.17 se transforma en un circuito síncrono añadiendo dos puertas AND que tienen como entradas x_1, x_2 y la señal de reloj. Las salidas de las dos puertas AND están a 0 mientras la señal de reloj vale 0. Asimismo se representa la salida de ambos tipos de circuitos cuando las señales de entrada cambian su valor a lo largo del tiempo de forma asíncrona.

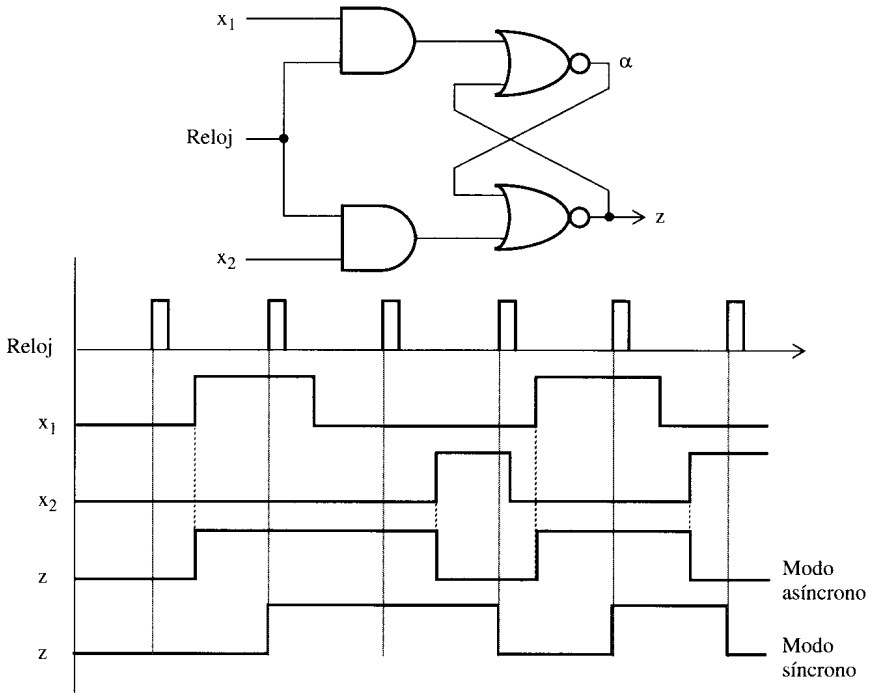


Figura B.21: Elemento de memoria síncrono

Como puede apreciarse en la Figura B.21, en el modo síncrono las señales x_1, x_2 y z no se muestrean arbitrariamente sino en los instantes en que la señal de reloj pasa de valor "0" a "1" (flanco de subida). En ese momento se determina el valor de z para un intervalo de tiempo que va hasta que se produzca el siguiente flanco de subida.

A veces puede interesar que , dado un elemento de memoria síncrono sea posible actuar también sobre su estado de una forma asíncrona. Para conseguirlo se dispone de dos entradas asíncronas que son independientes del reloj. Estas señales deben predominar sobre las síncronas y no activarse simultáneamente. La línea de puesta a "1" se conoce como *Preset* (*Pr*) y la de puesta a "0" como *Clear* (*Clr*).

En la Figura B.22 se muestra el elemento de memoria síncrono de la Figura B.21 con entradas adicionles de *Preset* y *Clear*. Puede observarse que cuando $Pr = 1$, el elemento de memoria se pone a "1" in-

dependientemente de los valores de x_1, x_2 y la señal de reloj; mientras que $Clr = 1$ lo pone a "0". La combinación $Pr = Clr = 1$ está prohibida porque origina el mismo tipo de indeterminación que cuando $x_1 = x_2 = 1$.

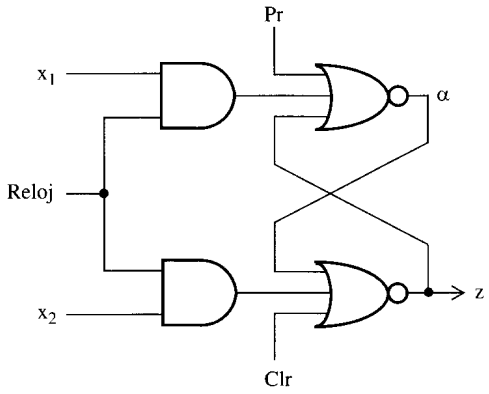


Figura B.22: Elemento de memoria síncrono con entradas asincronas de Preset y Clear

Sección B-3

Tipos de elementos de memoria

En la sección anterior se ha visto que un elemento de memoria o flip-flop es un circuito secuencial que tiene dos estados y que recuerda alguna característica de las señales de entrada. Como cualquier sistema secuencial viene descrito por sus funciones de transición de estado y de salida. Si se hace corresponder la salida con el estado solo es necesario especificar la función de transición para definir a un elemento de memoria. Esta función se puede dar como una tabla de estados, un diagrama de estados o una expresión lógica conocida como *función propia o característica* del elemento de memoria.

Un elemento de memoria (ver Figura B.23) tiene una o más entradas binarias y normalmente dos salidas que corresponden a la variable de estado (representada por Q) y su complemento ($\overline{Q}$). Cuando $Q = 1$ se dice que el elemento de memoria está en el "*estado 1*", y si $Q = 0$ en el "*estado 0*". En los elementos de memoria síncronos la entrada *Ck* (abreviatura de reloj en inglés (clock)) recibe la señal de reloj.

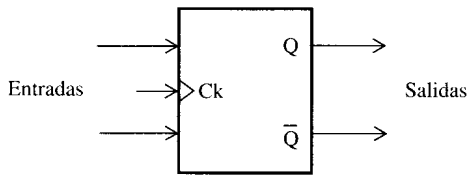


Figura B.23: Símbolo de un elemento de memoria

En esta sección se definen los tipos de elementos de memoria más utilizados en la práctica.

B.3.1 Elemento de memoria RS

Tiene dos entradas R (Reset o poner a 0) y S (Set o poner a 1). Es el elemento de memoria que se vio en la sección anterior ($x_1 = S$ y $x_2 = R$). En la Figura B.24 se muestra la tabla de estados, el diagrama de estados y la función característica del elemento de memoria *RS*.

La salida del elemento de memoria *RS* se pone a 1 si $S(t) = 1$ y $R(t) = 0$, se pone a 0 si $S(t) = 0$ y $R(t) = 1$, no cambia de valor cuando $S(t) = 0$ y $R(t) = 0$ y la combinación $S(t) = R(t) = 1$ no está permitida.

B.3.2 Elemento de memoria JK

Tiene dos entradas J y K y se puede considerar una generalización del elemento de memoria *RS* (siendo J equivalente a S y K equivalente a R) que elimina la indeterminación que produce la combinación prohibida.

Cuando $J(t) = K(t) = 1$ se produce un cambio de estado. En la Figura B.25 se muestra la tabla de estados, el diagrama de estados y la función característica del elemento de memoria *JK*.

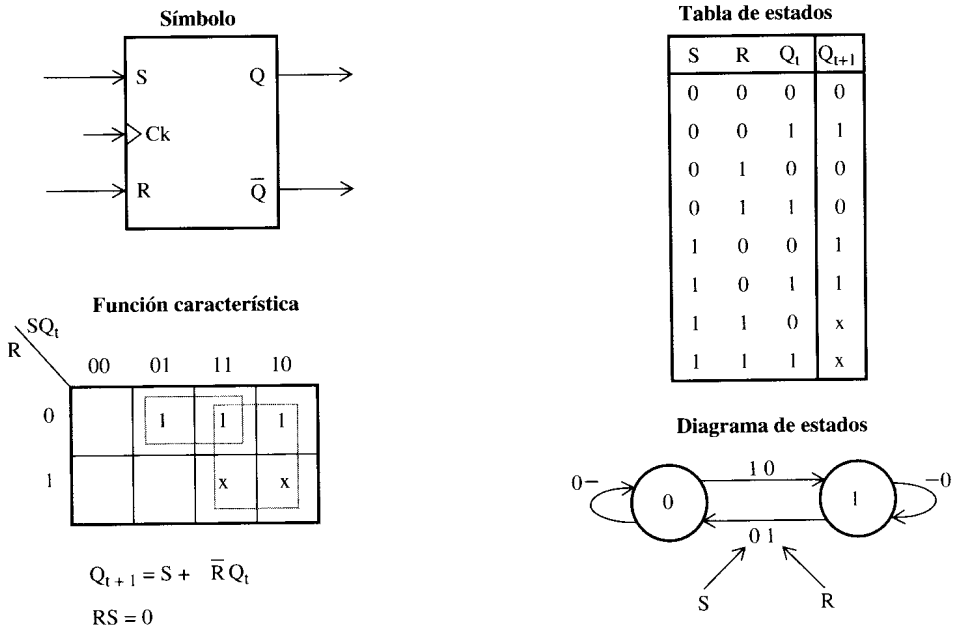


Figura B.24: Elemento de memoria *RS*

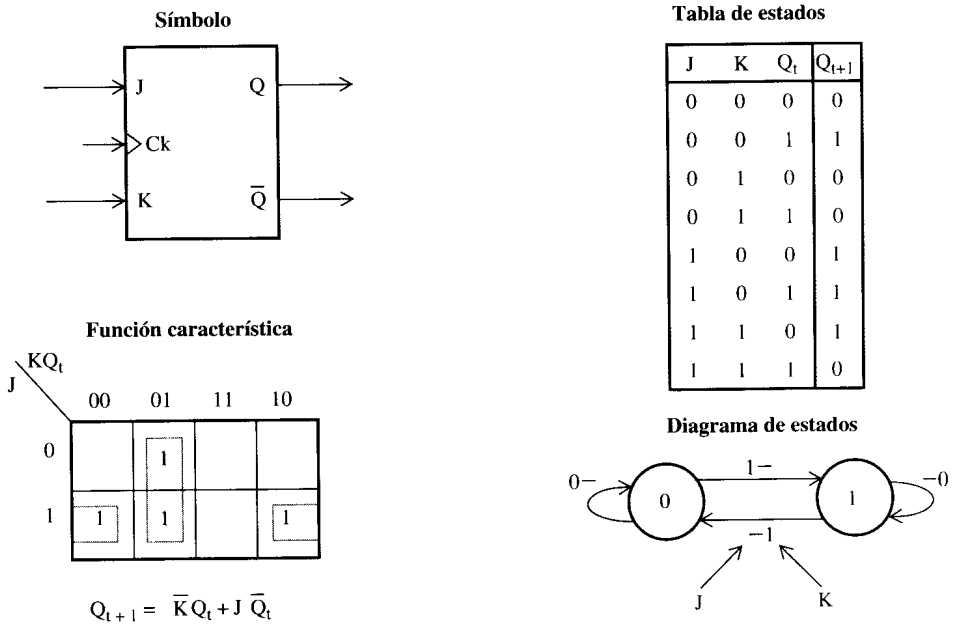


Figura B.25: Elemento de memoria *JK*

B.3.3 Elemento de memoria D

Tiene una entrada D (la D proviene del inglés "delay", en español "retardo"). El valor de la entrada $D(t)$ corresponde al próximo estado del elemento de memoria (ver Figura B.26).

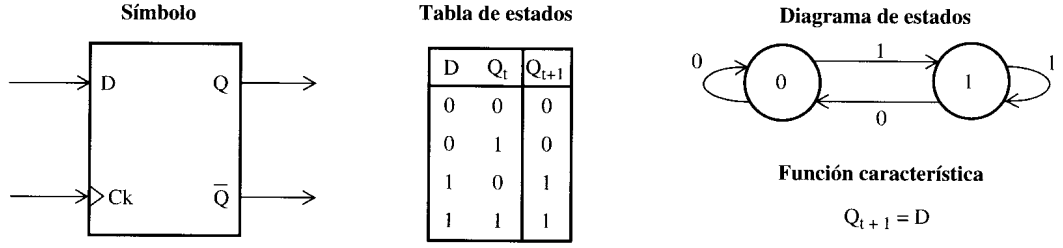


Figura B.26: Elemento de memoria D

Un elemento de memoria D se puede realizar a partir de un elemento de memoria RS o JK tal como se muestra en la Figura B.27.

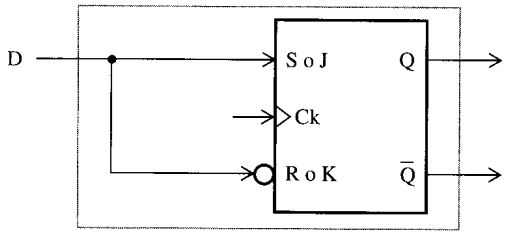


Figura B.27: Síntesis de un elemento de memoria D con un elemento de memoria RS o JK

Al existir un inversor a la entrada de R o K se observa que nunca pueden ser $R = S = 1$ y por lo tanto el elemento de memoria JK es equivalente en esta configuración al elemento de memoria RS.

B.3.4 Elemento de memoria T

Su nombre viene de la inicial de la palabra inglesa "toggle" que significa "conmutar". Tiene sólo una entrada T que cuando está a 1 hace cambiar la salida al estado contrario del que tuviese (ver Figura B.28).

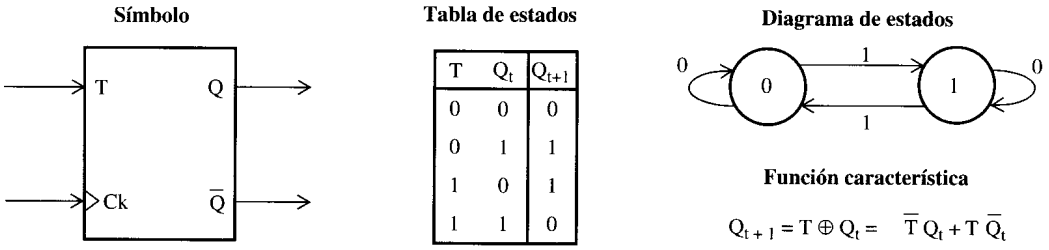


Figura B.28: Elemento de memoria T

El elemento de memoria *T* se puede obtener fácilmente del elemento de memoria *JK*. Si se comparan sus funciones características:

$$\text{JK:} \quad Q_{t+1} = \overline{K}\, Q_t + J\overline{Q}_t$$

$$\text{T:} \quad Q_{t+1} = \overline{T}\, Q_t + T\overline{Q}_t$$

se observa que si se hace $J = K = T$ se obtiene la misma expresión (ver Figura B.29)

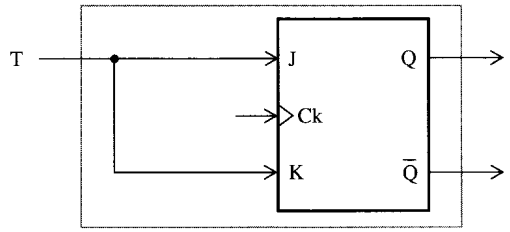


Figura B.29: Realización de un elemento de memoria *T* con un elemento de memoria *JK*

Sección B-4

Elemento de memoria maestro-esclavo

Se ha visto que se puede conseguir un elemento de memoria síncrono mediante la introducción adecuada de una señal de reloj a través de dos puertas AND, de manera que sólo se modifica el estado cuando dicha señal vale 1. Si la misma señal de reloj se presenta a todos los elementos con memoria en un circuito, todos están forzados a cambir de estado en sincronismo. Conviene analizar esta última afirmación con cierto detalle.

Una cuestión que surge es la siguiente: ¿Cuánto tiempo deben permanecer los pulsos de reloj en el nivel 1? Desde un punto de vista conceptual se pueden considerar impulsos de reloj ideales de anchura cero. En la práctica no es posible generar y transmitir tales pulsos instantáneos ya que las puertas lógicas no pueden responder a ellos. Una posibilidad es que la anchura de los pulsos sea igual al tiempo requerido para conmutar el estado de un elemento de memoria. Sin embargo, los retardos internos de los elementos de memoria varían de unidad a unidad y habría por lo tanto que dar como duración de los pulsos el de la unidad más lenta.

La necesidad de que la duración de los pulsos de reloj exceda al tiempo de conmutación de un elemento de memoria impone serias limitaciones a su utilización. Para poner esto de manifiesto se considera el circuito de la Figura B.30 en el que la salida Q de un elemento de memoria RS síncrono se conecta a su entrada.

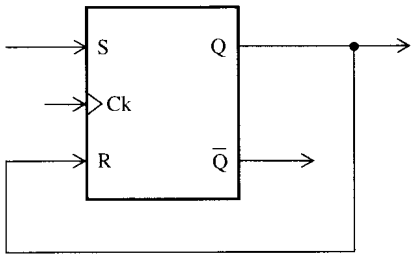


Figura B.30: Realimentación de la salida de un elemento de memoria RS

Se ha colocado un camino de realimentación con un retardo muy pequeño alrededor del elemento de memoria RS. Se supone que justamente antes de que la señal de reloj vaya al nivel 1, $Q = 0$ y $S = 1$. Cuando el reloj va al nivel 1, el elemento de memoria conmutará su estado $Q = 1 \Rightarrow R = 1$. Se tiene así que las entradas S y R son simultáneamente 1, de manera que la salida Q puede ir a 0 ó 1 y no se sabe de antemano en que estado se encontrará el elemento de memoria al llegar el siguiente pulso de reloj (ver Figura B.31).

En resumen, este ejemplo pone de manifiesto que *no se pueden utilizar fiablemente las señales de salida de un elemento de memoria para formar sus propias señales de entrada.* Sin embargo, existen muchas situaciones donde se precisa efectuar conexiones de este tipo.

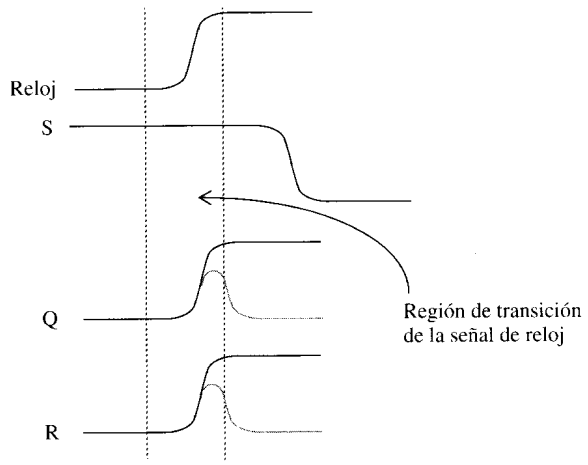


Figura B.31: Respuesta del circuito de la Figura B.30

Una solución al problema es utilizar relojes multifases (algunas señales de reloj de la misma frecuencia pero con fases diferentes). La Figura B.32 muestra un reloj bifásico y la forma en la que se puede emplear para conseguir una operación fiable. La salida del elemento de memoria controlado por la fase α se utiliza sólo para generar señales de entrada a los elementos de memoria que están conectados a la fase β y viceversa. Así, en la Figura B.32 la señal de entrada S debe venir generada a partir de un elemento de memoria de fase β. La diferencia de fase debe ser suficiente para permitir el cambio del primer elemento.

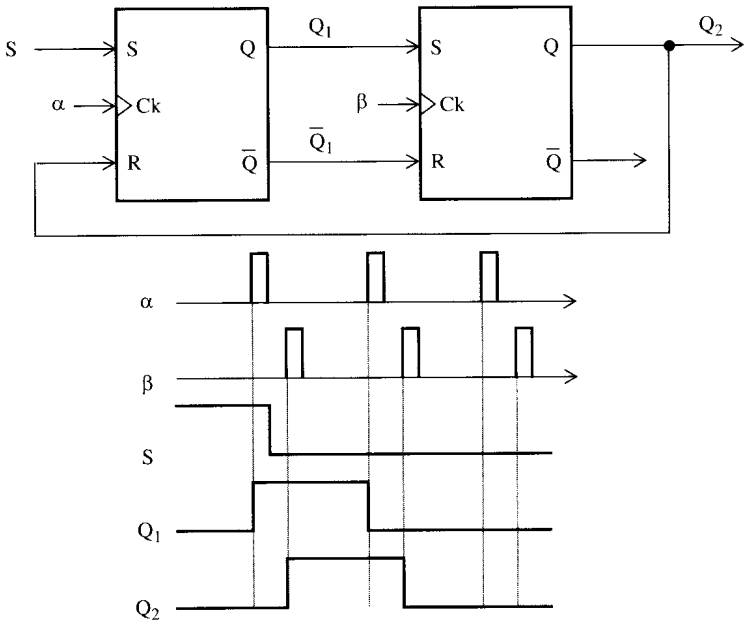


Figura B.32: Reloj bifásico para solventar el problema de realimentación de las salidas

La acción de incertidumbre del ejemplo anterior se elimina formando señales de entrada a los elementos de memoria que no varían cuando dichos elementos de memoria estan examinando sus señales de entrada.

Al aplicar este procedimiento un diseñador se puede equivocar y utilizar la señal de salida de un elemento de memoria de fase α como entrada a otro elemento de memoria también de fase α. Con frecuencia se deben introducir elementos de memoria adicionales para retardar una señal de manera que esté en su fase correcta. El método es pues engorroso y propenso a errores porque hay que mantener un estricto control de los diagramas de tiempo y de las fases de la señal de reloj de los distintos elementos de memoria.

Con la tecnología microelectrónica de hoy día resulta práctico construir un elemento de memoria a partir de un gran número de puertas lógicas. Los cambios en las señales de salida de tales elementos de memoria pueden retardarse mediante un esquema muy similar a la técnica del reloj bifásico.

En este caso se utilizan los flancos de subida y de bajada de los pulsos de reloj. El flanco de subida corresponde a la fase α, mientras que el de bajada sirve como fase β. La Figura B.33 muestra una estructura de este tipo que se conoce como elemento de memoria "*maestro-esclavo*" (flip-flop master-slave) que contiene dos elementos de memoria asíncronos dentro del mismo. El primero se denomina "*maestro*" mientras que el segundo es el "*esclavo*".

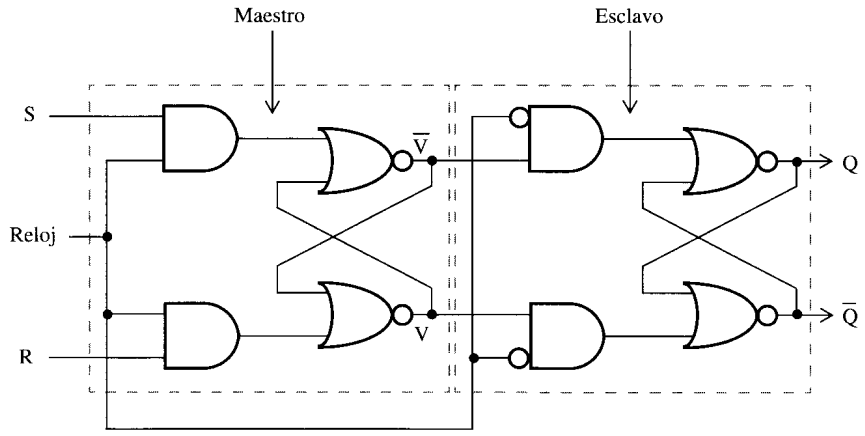


Figura B.33: Elemento de memoria *RS* de tipo maestro-esclavo

Cuando la señal de reloj toma el valor 1, el elemento de memoria maestro responde a las señales de entrada *S* y *R*. Estas señales no deben cambiar su valor durante la transición del reloj de 0 a 1 si se quiere que sea fiable el estado del maestro. En esta fase el elemento de memoria esclavo está desconectado del maestro por el segundo par de puertas AND, de manera que el valor de *Q* no cambia mientras el maestro está respondiendo a las señales de entrada. Al producirse el flanco de bajada del reloj (transición de 1 a 0) el maestro se desconecta de las señales de entrada externas y su estado (*V*) se transfiere al esclavo. Si *Q* cambia su valor ahora no se plantea ningún problema, porque el maestro al estar desconectado no está examinando sus señales de entrada y es posible por lo tanto realimentar las propias salidas de los elementos de memoria para formar sus señales de entrada.

En resumen, en los elementos de memoria tipo maestro-esclavo, las señales de entrada se muestrean durante el flanco de subida de la señal de reloj y sus salidas cambian con el flanco de bajada (ver Figura

B.34). Este modo de operación asegura que el elemento de memoria esclavo cambia de valor a lo más una vez durante cada período de reloj.

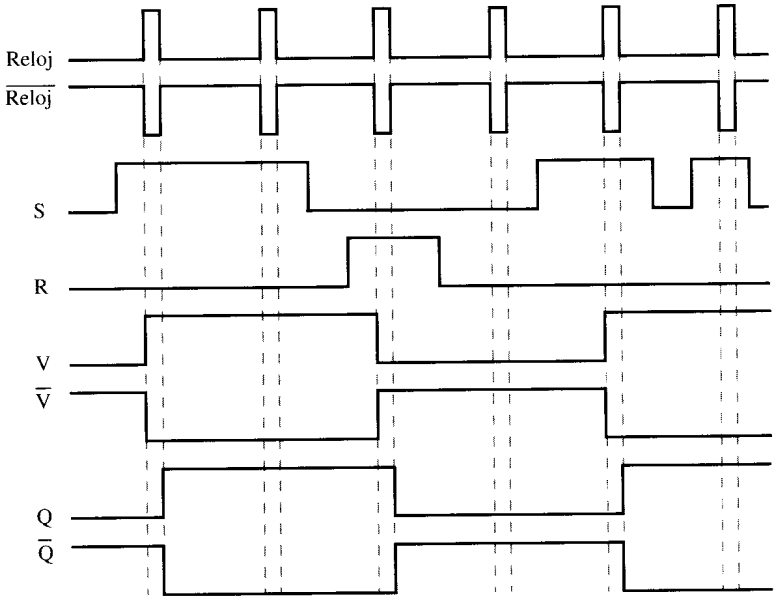


Figura B.34: Respuesta de un elemento de memoria *RS* de tipo maestro-esclavo

Sección B-5

Análisis de circuitos secuenciales

El problema de análisis consiste en determinar la descripción de un sistema secuencial del que se conoce su circuito. Se pueden distinguir dos casos:

a) Circuito secuencial realizado en forma canónica (ver Figura B.16).

b) Circuito secuencial realizado con elementos de memoria.

B.5.1 Circuito secuencial realizado en forma canónica

En este caso el proceso de análisis consta de los pasos siguientes:

1) Análisis de la parte combinacional del circuito. Esto permite determinar las expresiones lógicas de las funciones de transición f y de salida g del circuito secuencial dado.

2) Determinación de una especificación de alto nivel apropiada de estas funciones.

3) Descripción de la conducta temporal del sistema.

Un ejemplo sencillo muestra este método sistemático de análisis de circuitos secuenciales.

B.5.2 Ejemplo 1: Análisis de un circuito secuencial síncrono

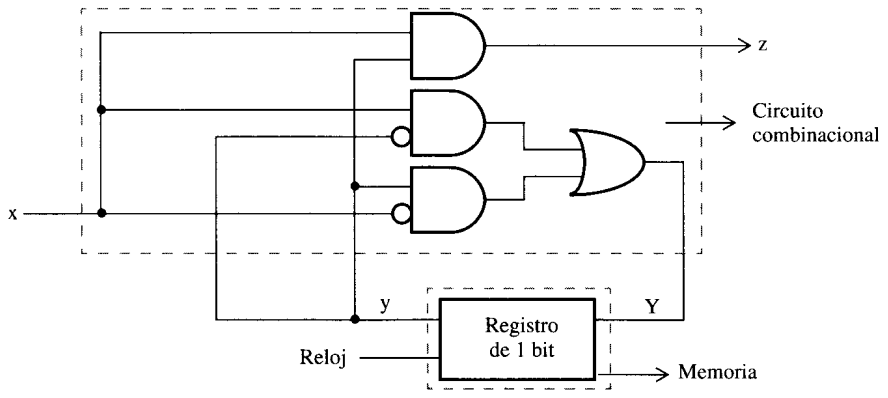


Figura B.35: Circuito secuencial del ejemplo B.5.2

El análisis de la parte combinacional del circuito de la Figura B.35 produce las expresiones siguientes para las funciones de transición de estados y de salida:

Función de transición de estado $f: Y = x\bar{y} + \bar{x}y$

Función de salida $g: z = xy$

Estas funciones se describen mediante la tabla de transición de estados y de salida que se muestra en la Figura B.36.

Estado presente y	Entrada	
	x = 0	x = 1
0	0 / 0	1 / 0
1	1 / 0	0 / 1
	Próximo estado / Salida	

Figura B.36: Tabla de transición de estados y de salida del Ejemplo B.5.2

Para obtener una descripción de alto nivel se realiza la siguiente codificación de la entrada, salida y estado (ver Figura B.37).

Codificación de la entrada

x	x
0	x_0
1	x_1

Codificación de la salida

z	z
0	z_0
1	z_1

Codificación del estado

y	s
0	S_0
1	S_1

Figura B.37: Codificación de la entrada, salida y estado del Ejemplo B.5.2

Sustituyendo la codificación de la Figura B.37 en la tabla de transición de estados, se obtiene la tabla de estados y el diagrama de estados de la Figura B.38.

Tabla de estados

Estado presente	Entrada	
	x_0	x_1
S_0	S_0 / z_0	S_1 / z_0
S_1	S_1 / z_0	S_0 / z_1
	Próximo estado / Salida	

Diagrama de estados

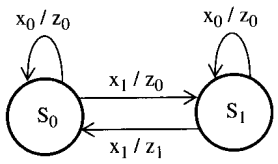


Figura B.38: Diagrama de estados y tabla de estados del ejemplo B.5.2

Del diagrama de estados de la Figura B.38 se puede deducir la siguiente especificación de alto nivel del circuito secuencial síncrono: "Una máquina secuencial posee una entrada x y una salida z y los conjuntos de

entrada y salida son respectivamente:

$$X = \{x_0, x_1\} \quad y \qquad Z = \{z_0, z_1\}$$

La salida en el instante de tiempo t vale z_1 si la cadena de símbolos de entrada que se ha recibido hasta ese instante contiene un número par de x_1's y z_0 en cualquier otro caso".

En la Figura B.39 se muestra una secuencia de pares de símbolos de entrada-salida, suponiendo que el estado inicial es S_0. Si el estado inicial fuese S_1 la especificación sería idéntica salvo que en lugar de detectar un número par de x_1's sería un número impar del mismo símbolo.

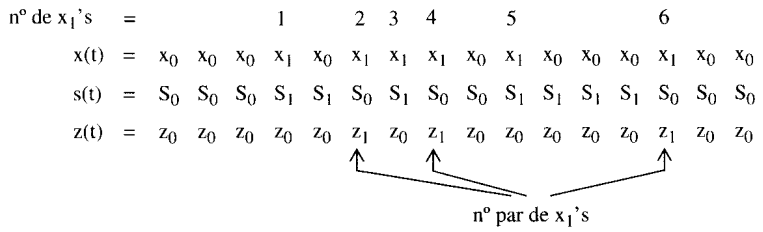


Figura B.39: Una secuencia de pares de símbolos de entrada-salida para el Ejemplo B.5.2

B.5.3 Circuito secuencial realizado con elementos de memoria

El análisis de un circuito secuencial realizado con elementos de memoria síncronos es una generalización del que se ha presentado para la realización canónica. En este caso la función de transición de estados del circuito secuencial se obtiene mediante el siguiente procedimiento:

1) Determinar las entradas a los elementos de memoria en función del estado presente y de las entradas externas al circuito. Esto requiere el análisis del correspondiente circuito combinacional.
2) Utilizar la función de transición o función característica de los elementos de memoria para calcular el próximo estado.
3) La función de salida se obtiene, de forma análoga al caso de la realización canónica, analizando el circuito combinacional correspondiente.

El ejemplo siguiente muestra el análisis de un circuito secuencial realizado con elementos de memoria síncronos.

B.5.4 Ejemplo 2: Análisis de un circuito secuencial síncrono

Sea el circuito secuencial de la Figura B.40. Las dos variables de estado son Q_1 y Q_2. Las funciones de entrada a los elementos de memoria *JK* vienen descritas por las siguientes expresiones lógicas:

$$J_1 = x_1 \qquad K_1 = \bar{x}_1$$

$$J_2 = x_2 z \qquad K_2 = \bar{z}$$

Si se tiene en cuenta la función característica del elemento de memoria *JK*:

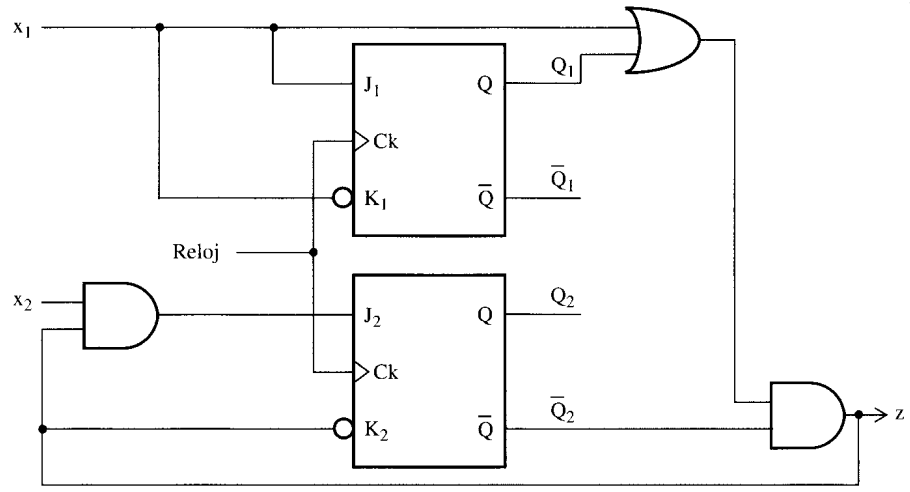


Figura B.40: Circuito secuencial del ejemplo B.5.4

$$Q(t+1) = \overline{K}\, Q(t) + J\, \overline{Q}\,(t)$$

las funciones de transición de estadosy de salida se puede expresar como:

$$Q_1(t+1) = Q_1(t)\, x_1 + \overline{Q}_1(t)\, x_1 = x_1$$

$$Q_2(t+1) = Q_2(t)\, z + \overline{Q}_2(t)\, x_2\, z = (Q_2(t) + \overline{Q}_2(t)\, x_2)\,(x_1 + Q_1(t))\, \overline{Q}_2\,(t) = x_1\, \overline{Q}_2(t) + Q_1(t))\, \overline{Q}_2\,(t)$$

$$z = (x_1 + Q_1(t))\, \overline{Q}_2\,(t)$$

La función de salida z se ha sustituido en $Q_2(t+1)$ para expresar el próximo estado $(Q_1(t+1), Q_2(t+1))$ solo en función del estado presente $(Q_1(t), Q_2(t))$ y de las entradas $(x_1(t), x_2(t))$. De estas expresiones se obtiene la tabla de transición de estados y de salida de la Figura B.41.

Estado presente $Q_1\, Q_2$	Entrada $x_1\, x_2$			
	0 0	0 1	1 0	1 1
0 0	0 0 / 0	0 0 / 0	1 0 / 1	1 1 / 1
0 1	0 0 / 0	0 0 / 0	1 0 / 0	1 0 / 0
1 0	0 0 / 1	0 1 / 1	1 0 / 1	1 1 / 1
1 1	0 0 / 0	0 0 / 0	1 0 / 0	1 0 / 0
	Próximo estado / salida			

Figura B.41: Tabla de transición de estados y de salida del ejemplo B.5.4

Por ejemplo cuando $(Q_1(t) = 1, Q_2(t) = 0)$ y $(x_1 = 0, x_2 = 1) \Rightarrow z = 1$ y $(Q_1(t+1) = 0, Q_2(t+1) = 1)$.

Para obtener una descripción de alto nivel se realiza la siguiente codificación de la entrada, salida y estado (ver Figura B.42).

Codificación de la entrada

x_1x_2	x
0 0	a
0 0	b
0 0	c
0 0	d

Codificación de la salida

z	z
0	z_0
1	z_1

Codificación del estado

Q_1Q_2	s
0 0	S_0
0 0	S_1
0 0	S_2
0 0	S_3

Figura B.42: Codificación de la entrada, salida y estado del ejemplo B.5.4

En la Figura B.43 se muestra la tabla de estados y el diagrama de estados del circuito secuencial que se deduce al sustituir la codificación de la Figura B.42 en la tabla de transición de estados.

Tabla de estados

Estado presente	Entrada			
	a	b	c	d
S_0	S_0 / z_0	S_0 / z_0	S_2 / z_1	S_3 / z_1
S_1	S_0 / z_0	S_0 / z_0	S_2 / z_0	S_2 / z_0
S_2	S_0 / z_1	S_1 / z_1	S_2 / z_1	S_3 / z_1
S_3	S_0 / z_0	S_0 / z_0	S_2 / z_0	S_2 / z_0
	Próximo estado / salida			

Diagrama de estados

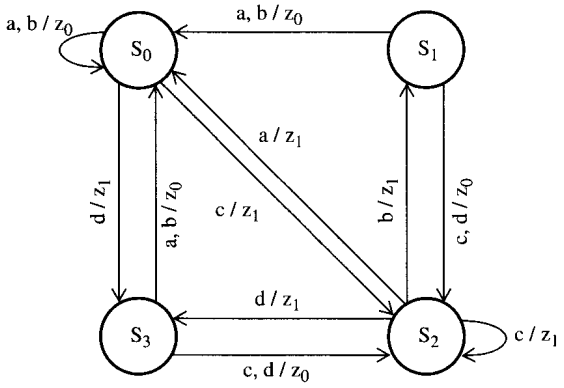


Figura B.43: Diagrama de estados y tabla de estados del ejemplo B.5.4

En la Figura B.44 se resume el proceso de análisis de un circuito secuencial síncrono que sigue la secuencia de pasos siguiente:

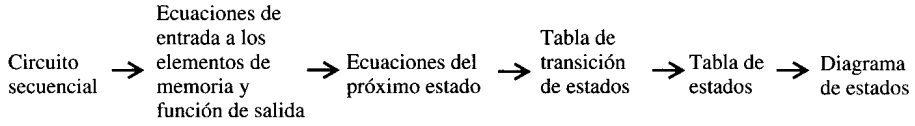


Figura B.44: Procedimiento de análisis de un circuito secuencial

Sección B-6

Síntesis de circuitos secuenciales

En el problema de diseño el objetivo es obtener un circuito secuencial a partir de las especificaciones de una conducta secuencial deseada. Es por lo tanto el problema inverso del de análisis que se ha tratado en la sección anterior. En la Figura B.45 se muestran los pasos que hay que dar en cualquier problema de síntesis.

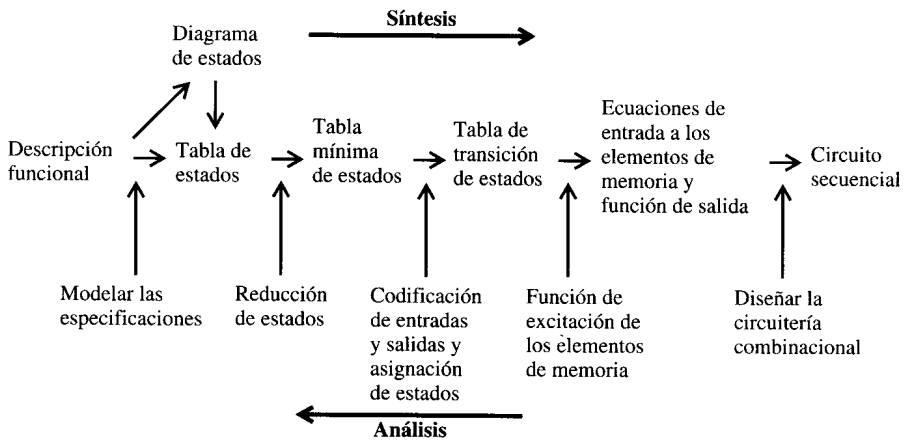


Figura B.45: Procedimiento de síntesis de un circuito secuencial

Un método sistemático de síntesis de circuitos secuenciales se realiza en los pasos siguientes:

1) Modelar las especificaciones de la conducta secuencial deseada.
2) Minimizar el número de estados del modelo obtenido en el paso anterior.
3) Codificar las entradas, salidas y estados (problema de la asignación de estados).
4) Seleccionar el tipo de elementos de memoria que se va a utilizar y determinar las ecuaciones de entrada a los elementos de memoria.
5) Diseñar la circuitería combinacional.

B.6.1 Modelar las especificaciones

El objetivo de este primer paso es intentar determinar una máquina secuencial $M = (X, Z, S, f, g)$ que cumpla las especificaciones del problema. Es el punto más importante de todo el procedimiento y el más difícil de

automatizar, ya que se trata de traducir una descripción expresada en lenguaje natural en una estructura formal que la represente de de forma exacta. A partir de aquí todo el proceso se convierte en una técnica de naturaleza mecánica sin ningún tipo de dificultad intrínseca.

Pueden existir muchas máquinas secuenciales que satisfagan las especificaciones, sin embargo, en este punto el objetivo es encontrar al menos una que las cumpla y dar su descripción mediante un diagrama de estados o una tabla de estados. Las máquinas secuenciales de estados finitos admiten una cierta clasificación. Por ejemplo, muchas máquinas tienen un estado (o conjunto de estados) a través del cual pasa periódicamente sin tener en cuenta la secuencia de símbolos de entrada que haya recibido. A menudo es natural comenzar el proceso de determinación de la máquina secuencial suponiendo que comienza su funcionamiento en este estado que se denomina *estado inicial.*

Otra propiedad importante que exhiben cierto tipo de máquinas es que además de poseer un estado inicial tratan únicamente con *secuencias de entrada de longitud fija.* Por ejemplo reconocer secuencias binarias de 3 bits que poseen 2 unos. Si la longitud de estas secuencias es λ y el número de símbolos de entrada es N, sólo necesitan considerarse N^{λ} secuencias por lo que puede asociarse una secuencia de estados con cada una de ellas. La forma de modelar las especificaciones es mediante un diagrama de estados o una tabla de estados, que deben de reconocer todas las condiciones o situaciones que pueden presentarse a la máquina y a las que ésta tiene que responder.

La segunda categoría de máquinas difiere de la primera porque las secuencias que reconoce no tienen una longitud fija. El modelo de la caja de caudales de la Figura B.1 pertenece a esta categoría de máquinas. La categoría final de máquinas de estados finito consiste en aquellas en las que no existen estados a través de los cuales la máquina deba necesariamente pasar. Es posible referirse a un cierto estado como estado inicial, en el sentido de que la máquina comienza a operar en ese estado pero no necesita nunca retornar a él.

B.6.2 Minimizar el número de estados

Generalmente la primera descripción que se hace de una máquina contendrá *estados equivalentes.* De manera informal, se dice que dos estados son equivalentes si es imposible distinguirlos observando las secuencias de salida que se producen al enviar a la máquina cualquier secuencia de entrada. Es muy deseable, desde el punto de vista de coste del circuito, eliminar la duplicación que representan los estados equivalentes. En el apartado B.1.4 se presentó un procedimiento algorítmico para minimizar el número de estados (método de las particiones).

B.6.3 Codificar en binario las entradas, salidas y estado

El problema de la codificación binaria de una máquina secuencial se presentó en el apartado B.1.5 y por lo tanto no será repetido aquí.

B.6.4 Ecuaciones de entrada a los elementos de memoria

El vector binario de estado proveniente de la asignación de estados tiene que almacenarse en un conjunto de elementos de memoria. Es necesario seleccionar el tipo de elemento de memoria que se va a utilizar. Aunque en la mayoría de los casos todos los elementos de memoria de un circuito secuencial son del mismo tipo, el procedimiento se puede aplicar también al caso en que sean diferentes.

De la tabla de estados se deduce la *tabla de transición de estados* al sustituir cada estado por su correspondiente asignación. La tabla de transición de estados especifica la función de transición de estados f

que da para cada combinación de entradas binarias (x_1, x_2, ..., x_n) y estados presentes ($Q_1(t)$, $Q_2(t)$, ..., $Q_k(t)$) cual es el próximo estado de los elementos de memoria ($Q_1(t+1)$, $Q_2(t+1)$, ..., $Q_k(t+1)$). Para generar estas $Q_i(t+1)$ hay que aplicar a los elementos de memoria las entradas adecuadas.

Las funciones lógicas que describen el efecto de las entradas del circuito x_1, x_2, ..., x_n y el estado presente $Q_1(t)$, $Q_2(t)$, ..., $Q_k(t)$ sobre las entradas a los elementos de memoria se conocen como *funciones de excitación*. Estas funciones se determinan a partir de las *tablas de excitación* de los diferentes elementos de memoria, que especifican las entradas que se le tienen que aplicar para producir una transición dada. En la Figura B.46 se resumen las tablas de excitación de los diferentes elementos de memoria.

Tablas de excitación de los elementos de memoria RS, JK, D y T

Q_t	Q_{t+1}	S	R	J	K	D	T
0	0	0	-	0	-	0	0
0	1	1	0	1	-	1	1
1	0	0	1	-	1	0	1
1	1	-	0	-	0	1	0

$D(t) = Q(t+1)$

$T(t) = Q(t) \oplus Q(t+1)$

Figura B.46: Tablas de excitación de los distintos tipos de elementos de memoria

Estas tablas se deducen directamente de las tablas de estados de los elementos de memoria dadas en la sección B-3. El símbolo - en las tablas de excitación denota una situación donde el valor de la entrada al elemento de memoria es inoperante ya que no afecta al valor de la salida. En el caso de los elementos de memoria *D* y *T* es posible dar una expresión lógica para sus funciones de excitación (ver Figura B.46). Debido a los términos inoperantes no es conveniente utilizar expresiones para los elementos de memoria *RS* y *JK*.

B.6.5 Ejemplo 1: Detector de secuencia mediante un modelo de Huffman-Mealy

Especificación del problema

Diseñar un circuito secuencial con elementos de memoria tipo *D*, que posee una entrada *x* y una salida *z* y los conjuntos de entrada y salida son respectivamente:

$$X = \{0, 1\} \quad y \quad Z = \{0, 1\}$$

La salida en el instante de tiempo *t* vale 1 siempre que en la entrada se detecta la secuencia 0101. Por ejemplo, cuando la secuencia de entrada es $x = 010101$ la correspondiente secuencia de salida es 000101. De manera más formal se puede expresar como:

$$z(t) = 1 \quad \text{si} \quad x(t-3, t) = 0101$$
$$z(t) = 0 \quad \text{en cualquier otro caso}$$

Paso 1 Modelización de las especificaciones

Para comenzar a modelar las especificaciones del problema, se supone que en el instante t_0 la máquina se encuentra en el estado inicial S_0. Mientras se encuentra en este estado la máquina puede recibir una entrada de un 0 ó un 1. Para cada una de estas entradas se dibuja un arco que parte desde S_0 y termina en el próximo estado apropiado (ver Figura B.47). El arco etiquetado 1/0 forma un bucle alrededor del estado S_0, puesto que

la máquina no inicia el proceso de detección de la secuencia hasta que recibe una entrada que vale 0. Una entrada de 0 indica un posible comienzo de la secuencia a detectar y por lo tanto se dibuja un arco etiquetado 0/0 que va desde S_0 a S_1. Cuando la máquina está en el estado S_1, una entrada igual a 1 la lleva al estado S_2 mientras que una entrada de 0 la deja en el mismo estado. Si en el estado S_2 se recibe un 1 las dos últimas entradas habrán sido 11 y como esta secuencia de entrada no se puede completar de ninguna forma para dar 0101, la máquina se dirige otra vez al estado inicial S_0. Al estado S_3 se llega después de haber recibido una secuencia de entrada cuyos tres últimos valores son 010. Una entrada adicional de 1 produce una salida $z = 1$ (se ha detectado la secuencia pedida) y se origina una transición desde el estado S_3 hacia el estado S_2, que es el estado que corresponde a secuencias de entrada cuyos dos últimos símbolos han sido 01. Una entrada de 0 aplicada a la máquina cuando está en S_3 la transfiere al estado S_1, puesto que este valor puede ser el comienzo de una nueva secuencia 0101.

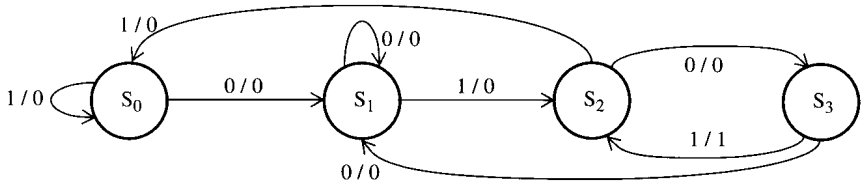


Figura B.47: Diagrama de estados del modelo de Huffman-Mealy del detector de secuencias 0101

En resumen, la máquina está en el estado S_i, ($i = 0, 1, 2, 3$) si:

S_0	estado inicial
S_1	$x(t) = 0$
S_2	$x(t-1, t) = 01$
S_3	$x(t-2, t) = 010$

En la Figura B.48 se muestra la tabla de estados correspondiente.

Estado presente	Entrada 0	Entrada 1
S_0	S_1 / 0	S_0 / 0
S_1	S_1 / 0	S_2 / 0
S_2	S_3 / 0	S_0 / 0
S_3	S_1 / 0	S_2 / 1
	Próximo estado / salida	

Figura B.48: Tabla de estados del modelo de Huffman-Mealy del detector de secuencias 0101

Paso 2 Minimización del número de estados

Si se aplica el método de las particiones a la tabla de estados de la Figura B.48 es fácil deducir que no es posible ninguna reducción en el número de estados.

Paso 3 Asignación de estados

Se necesitan dos variables de estado con $2^2 = 4$ estados para la representación de los cuatro estados del detector de secuencia. En la Figura B.49 se muestran dos posibles asignaciones de estados.

1ª Asignación

	Q_1 Q_2
$S_0 \rightarrow$	0 0
$S_1 \rightarrow$	0 1
$S_2 \rightarrow$	1 1
$S_3 \rightarrow$	1 0

2ª Asignación

	Q_1 Q_2
$S_0 \rightarrow$	0 0
$S_1 \rightarrow$	0 1
$S_2 \rightarrow$	1 0
$S_3 \rightarrow$	1 1

Figura B.49: Dos posibles asignaciones de estado para el detector de secuencias

Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.50 se muestran las tablas de transición de estados y de salida que se obtienen al sustituir las dos asignaciones de estados de la Figura B.49 en la tabla de estados de la Figura B.48.

1ª Asignación

Estado presente Q_1 Q_2	Entrada x	
	0	1
0 0	0 1 / 0	0 0 / 0
0 1	0 1 / 0	1 1 / 0
1 1	1 0 / 0	0 0 / 0
1 0	0 1 / 0	1 1 / 1
	Próximo estado / salida	

2ª Asignación

Estado presente Q_1 Q_2	Entrada x	
	0	1
0 0	0 1 / 0	0 0 / 0
0 1	0 1 / 0	1 0 / 0
1 0	1 1 / 0	0 0 / 0
1 1	0 1 / 0	1 0 / 1
	Próximo estado / salida	

Figura B.50: Tablas de transición de estados y de salida para el detector de secuencias

Teniendo en cuenta la función de excitación de un elemento de memoria D:

$$D(t) = Q(t + 1)$$

y las tablas de transición de estados de la Figura B.50, se deducen las expresiones de las entradas D_1 y D_2 a los dos elementos de memoria para las dos asignaciones de estados (ver Figura B.51).

Paso 5 Realización del circuito secuencial

En la Figura B.52 se muestra la realización del circuito secuencial del detector de secuencias para las dos asignaciones de estados realizadas. Se observa como la segunda asignación que se deduce de la primera asignación, cambiando simplemente los códigos de los estados S_2 y S_3, requiere menos de la mitad de puertas lógicas para sintetizar la parte combinacional del circuito. Como ya se indicó, no existe ningún procedimiento sistemático que permita obtener, dado un diagrama de estados, cual es la asignación de estados que da lugar a que la parte combinacional del circuito secuencial resultante sea de coste mínimo.

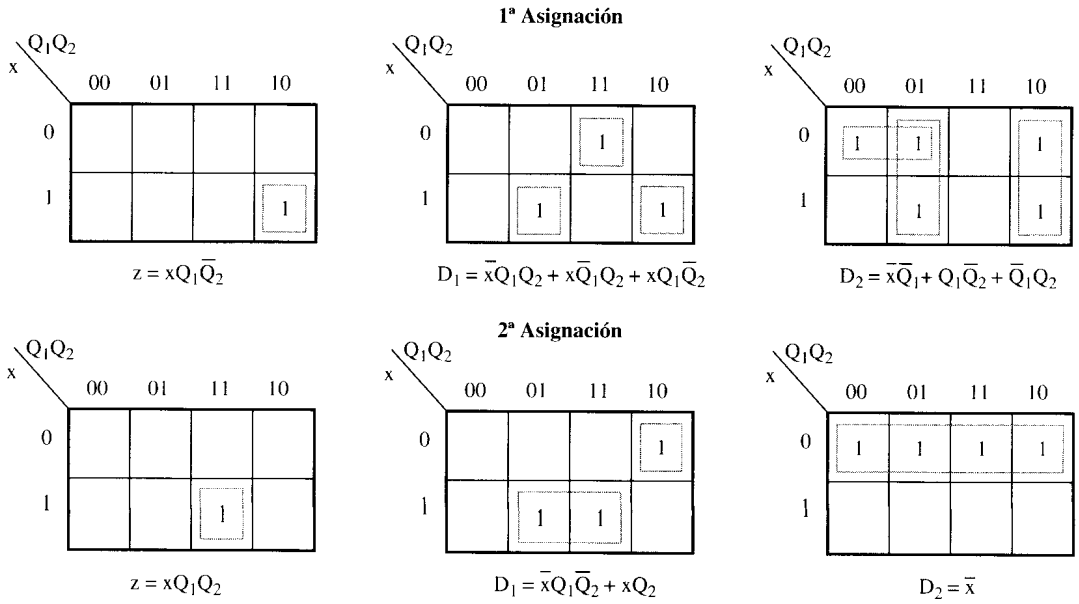


Figura B.51: Funciones de excitación y salida del detector de secuencias

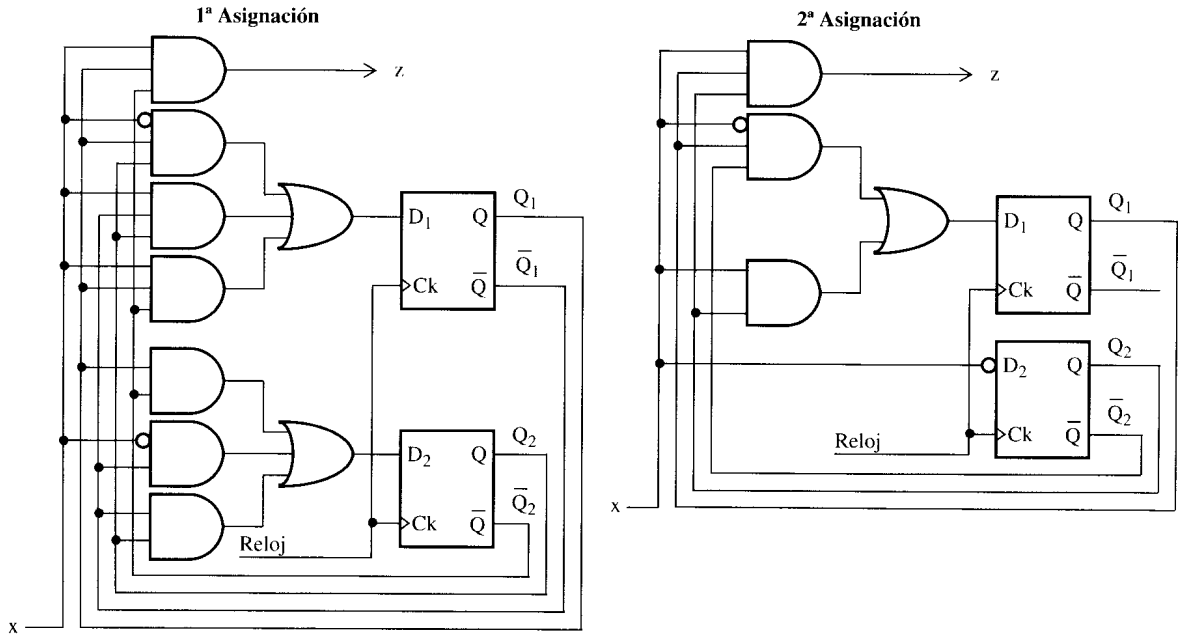


Figura B.52: Circuito secuencial del detector de secuencias

B.6.6 Ejemplo 2: Detector de secuencia mediante un modelo de Moore

Especificación del problema

Son las mismas del ejemplo 1: la salida en el instante de tiempo t vale 1 siempre que en la entrada se detecta la secuencia 0101.

Paso 1 Modelización de las especificaciones

En la Figura B.53 se muestra el diagrama de estados que describe las especificaciones del problema mediante un modelo de Moore.

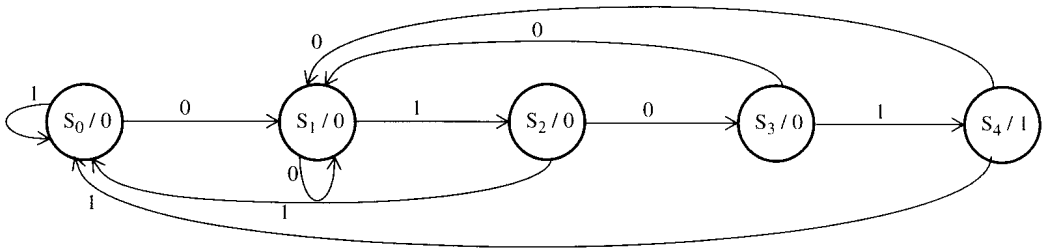


Figura B.53: Diagrama de estados del modelo de Moore del detector de secuencias 0101

En la Figura B.54 se muestra la tabla de estados correspondiente.

Estado presente	Entrada x = 0	Entrada x = 1	Salida
S_0	S_1	S_0	0
S_1	S_1	S_2	0
S_2	S_3	S_0	0
S_3	S_1	S_4	0
S_4	S_1	S_0	1
	Próximo estado		

Figura B.54: Tabla de estados del modelo de Moore del detector de secuencias 0101

Paso 2 Minimización del número de estados

Si se aplica el método de las particiones a la tabla de estados de la Figura B.54 es fácil deducir que no es posible ninguna reducción en el número de estados.

Paso 3 Asignación de estados

Se necesitan tres variables de estado para la representación de los cinco estados del detector de secuencia. El modelo de Moore necesita un estado más que el de Huffman-Mealy. En la Figura B.55 se aplica la Regla 1 de asignación de estados del apartado B.1.5 (hay que asignar códigos que difieren en una variable a los estados presentes que llevan al sistema al mismo próximo estado) a la tabla de estados de la Figura B.54.

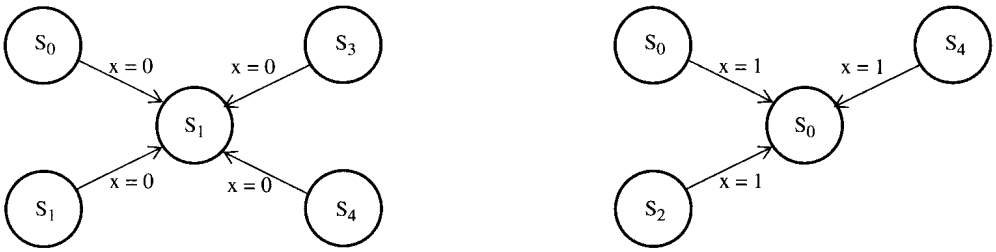


Figura B.55: Aplicación de la Regla 1 al ejemplo de la Figura B.54

De la Figura B.55 se sigue que de una parte los estados (S_0, S_1, S_3, S_4) y de otra (S_0, S_2, S_4) deben tener codificaciones adyacentes. Para este ejemplo sólo con esta regla queda completamente determinada la asignación de estados. En la Figura B.56 se muestra una posible asignación de estados.

	Q_1	Q_2	Q_3
$S_0 \rightarrow$	0	0	0
$S_1 \rightarrow$	0	0	1
$S_2 \rightarrow$	0	1	0
$S_3 \rightarrow$	1	0	1
$S_4 \rightarrow$	1	0	0

Figura B.56: Asignación de estado para el detector de secuencias

Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.57 se muestran las tablas de transición de estados y de salida que se obtienen al sustituir la asignación de estados de la Figura B.56 en la tabla de estados de la Figura B.54.

Estado presente	Entrada x		Salida
Q_1 Q_2 Q_3	0	1	z
0 0 0	0 0 1	0 0 0	0
0 0 1	0 0 1	0 1 0	0
0 1 0	1 0 1	0 0 0	0
1 0 1	0 0 1	1 0 0	0
1 0 0	0 0 1	0 0 0	1
	Próximo estado		

Figura B.57: Tabla de transición de estados y de salida del detector de secuencias

Teniendo en cuenta la función de excitación de un elemento de memoria D ($D(t) = Q(t + 1)$)y las tablas de transición de estados de la Figura B.57, se deducen las expresiones de las entradas D_1, D_2 y D_3 a los tres elementos de memoria para la asignación de estados planteada (ver Figura B.58).

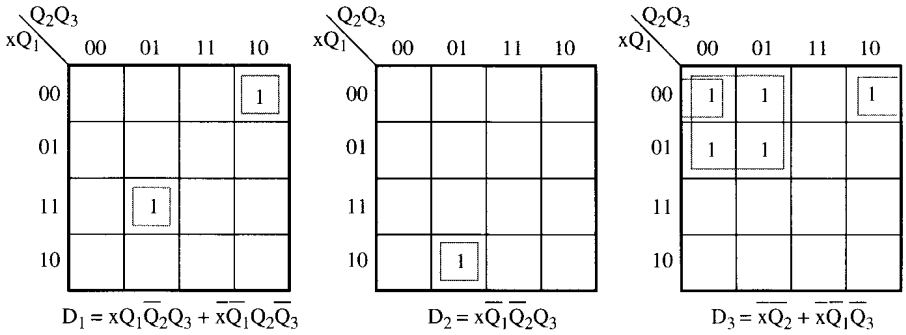


Figura B.58: Funciones de excitación del detector de secuencias

La función de salida (que en el modelo de Moore sólo es función del estado presente) se deduce directamente de la tabla de salida de la Figura B.57 y viene dada por: $z = Q_1\overline{Q}_2\overline{Q}_3$.

Paso 5 Realización del circuito secuencial

En la Figura B.59 se muestra la realización del circuito secuencial del detector de secuencias diseñado con el modelo de Moore para la asignación de estados realizada.

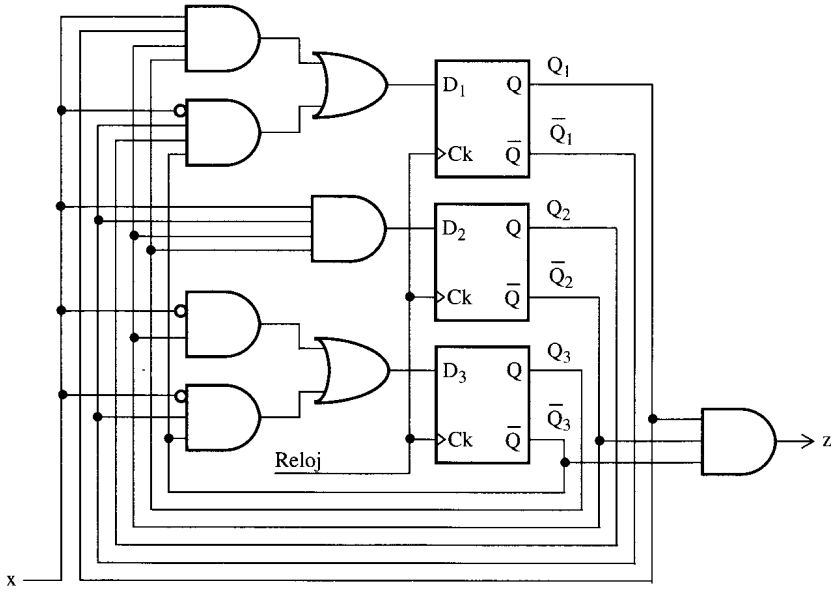


Figura B.59: Circuito secuencial del detector de secuencias

B.6.7 Ejemplo 3: Complementador a 2

Especificación del problema

Diseñar un circuito secuencial con elementos de memoria tipo *JK*, que posee una entrada x y una salida z y los conjuntos de entrada y salida son respectivamente:

$$X = \{0, 1\} \quad y \quad Z = \{0, 1\}$$

La salida z es el complemento a 2 de x. Por ejemplo, si la secuencia de entrada es x = 001101000 la correspondiente secuencia de salida es 110011000. Se supone que el bit situado más a la derecha (el menos significativo) es el primero en entrar en el circuito secuencial.

Paso 1 Modelización de las especificaciones

En el Tema 4 se expuso un método rápido de obtener en forma serie el complemento a 2 de un número binario. Consiste en ir examinando los bits del número de derecha a izquierda que se dejan inalterados hasta la aparición del primer 1, que se deja inalterado y a partir de ese instante se complementan todos los bits (ver Figura 4.48). La máquina secuencial comienza en el estado S_0 y permanece en ese estado dando salida 0 mientras las entradas que recibe son 0's. En el instante que recibe un 1 se transfiere al estado S_1 y da salida 1. Desde ese momento permanece en el estado S_1 y para cada entrada que le llega genera como salida su complemento. En la Figura B.60 se muestra el diagrama de estados y la tabla de estados correspondientes.

Diagrama de estados

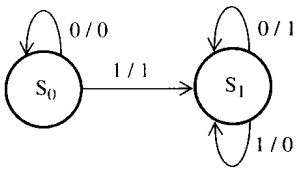


Tabla de estados

Estado presente	Entrada 0	Entrada 1
S_0	S_0 / 0	S_1 / 1
S_1	S_1 / 1	S_1 / 0
	Próximo estado / salida	

Figura B.60: Diagrama de estados y tabla de estados del complementador a 2

Paso 2 Minimización del número de estados

Resulta inmediato ver que no es posible reducir el número de estados de la tabla de estados de la Figura B.60 (S_0 y S_1 no son 1-equivalentes).

Paso 3 Asignación de estados

En el complementador a 2 se necesita una variable de estado Q_1 con $2^1 = 2$ estados. En la Figura B.61 se muestra la asignación de estados realizada.

Q_1
$S_0 \rightarrow 0$ $S_1 \rightarrow 1$

Figura B.61: Asignación de estados para el complementador a 2

Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.62 se da la tabla de transición de estados y de salida que se obtiene al sustituir la asignación de estados de la Figura B.61 en la tabla de estados de la Figura B.60.

Estado presente Q_1	Entrada	
	x = 0	x = 1
0	0 / 0	1 / 1
1	1 / 1	1 / 0
	Próximo estado / salida	

Figura B.62: Tabla de transición de estados y de salida del complementador a 2

Teniendo en cuenta la tabla de excitación de un elemento de memoria *JK* y la tabla de transición de estados de la Figura B.62 se deducen las expresiones lógicas de las entradas J_1 y K_1 al elemento de memoria. En la Figura B.63 se muestran los mapas de Karnaugh correspondientes a J_1, K_1 y z.

Tabla de excitación del elemento de memoria JK

Q_t	Q_{t+1}	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

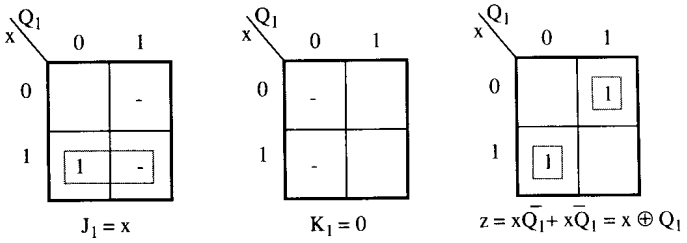


Figura B.63: Funciones de excitación y salida del complementador a 2

Paso 5 Realización del circuito secuencial

En la Figura B.64 se muestra la realización del circuito secuencial del complementador a 2 para la asignación de estados realizada.

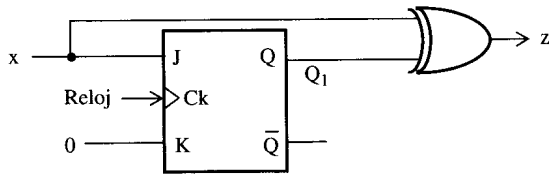


Figura B.64: Circuito secuencial del complementador a 2

B.6.8 Ejemplo 4: Contador binario módulo-8

Especificación del problema

Diseñar un circuito secuencial con elementos de memoria tipo *T*, que posee una entrada binaria *x* y una salida

z cuyos valores pertenecen al conjunto {0,1,2,3,4,5,6,7}. La salida cuenta módulo-8 el número de 1's, no necesariamente consecutivos, que hay en la secuencia de entrada (es decir 0,1,2,3,4,5,6,7,0,1,2,...).

Paso 1 Modelización de las especificaciones

En la Figura B.65 se da el diagrama de estados y la tabla de estados del contador binario módulo 8. Las transiciones entre estados ocurren si al llegar un pulso de reloj es $x = 1$. En este ejemplo el contador binario se ha modelado como una máquina secuencial tipo Moore.

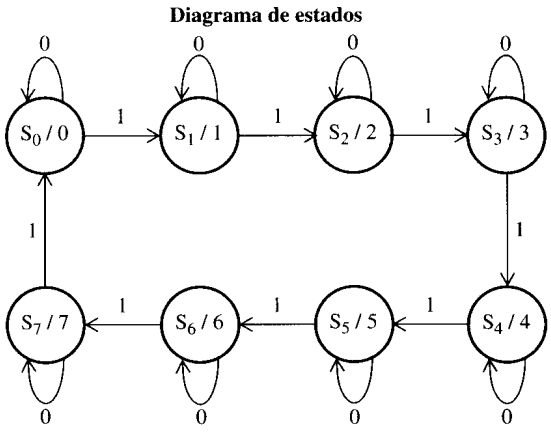


Tabla de estados

Estado presente	Entrada x = 0	Entrada x = 1	Salida
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_3	2
S_3	S_3	S_4	3
S_4	S_4	S_5	4
S_5	S_5	S_6	5
S_6	S_6	S_7	6
S_7	S_7	S_0	7
	Próximo estado		

Figura B.65: Diagrama de estados y tabla de estados del contador binario módulo-8

Paso 2 Minimización del número de estados

No es posible reducirlos (S_0, S_1, S_2, ..., S_7 no son 1-equivalentes).

Paso 3 Codificación de las salidas y asignación de estados

Para efectuar el diseño es preciso codificar las salidas. El conjunto de valores que toma la salida requiere al menos 3 variables binarias. Una posibilidad es codificar el valor decimal i por su representación binaria. Se necesitan 3 variables de estado con $2^3 = 8$ estados para la representación de los ocho estados del contador módulo-8. El circuito combinacional que genera la salida se puede eliminar si se utiliza para la asignación de estados la misma codificación de las salidas. En la Figura B.66 se muestra dicha codificación.

z	z_1	z_2	z_3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

	Q_1	Q_2	Q_3
$S_0 \rightarrow$	0	0	0
$S_1 \rightarrow$	0	0	1
$S_2 \rightarrow$	0	1	0
$S_3 \rightarrow$	0	1	1
$S_4 \rightarrow$	1	0	0
$S_5 \rightarrow$	1	0	1
$S_6 \rightarrow$	1	1	0
$S_7 \rightarrow$	1	1	1

Figura B.66: Codificación de las salidas y asignación de estados para el contador módulo-8

Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.67 se muestra la tabla de transición de estados que se obtiene al sustituir la asignación de estados de la Figura B.66 en la tabla de la Figura B.65. La salida $z(t)$ del contador módulo-8 coincide con el estado presente $Q(t) = (Q_1(t), Q_2(t), Q_3(t))$.

Estado presente $Q_1\ Q_2\ Q_3$	Entrada x	
	0	1
0 0 0	0 0 0	0 0 1
0 0 1	0 0 1	0 1 0
0 1 0	0 1 0	0 1 1
0 1 1	0 1 1	1 0 0
1 0 0	1 0 0	1 0 1
1 0 1	1 0 1	1 1 0
1 1 0	1 1 0	1 1 1
1 1 1	1 1 1	0 0 0
	Próximo estado	

Figura B.67: Tabla de transición de estados del contador módulo-8

Teniendo en cuenta la función de excitación de un elemento de memoria T:

$$T(t) = Q(t) \oplus Q(t+1)$$

y la tabla de transición de estados de la Figura B.67 se deducen las expresiones de las entradas T_1, T_2 y T_3 a los tres elementos de memoria (ver Figura B.68).

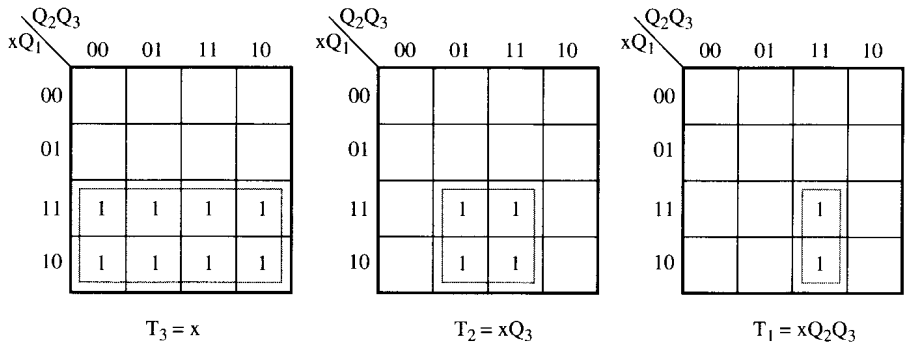


Figura B.68: Funciones de excitación del contador módulo-8

Paso 5 Realización del circuito secuencial

En la Figura B.69 se muestra la realización del circuito secuencial del contador módulo-8.

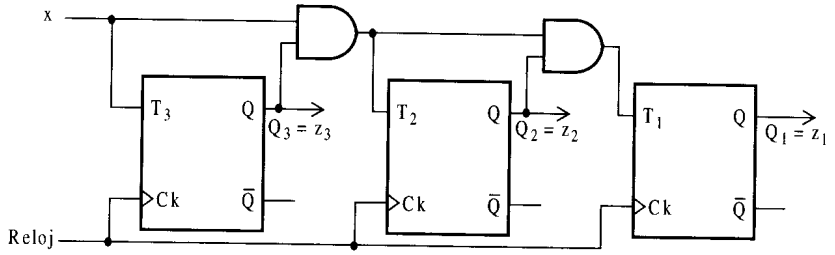


Figura B.69: Circuito secuencial del contador módulo-8

B.6.9 Ejemplo 5: Contador de código arbitrario con dos secuencias

Especificación del problema

Diseñar un contador síncrono que siga una de dos posibles secuencias de repetición dependiendo del valor de una entrada de control C. Cuando $C = 0$, la secuencia es 00, 01, 11 y cuando $C = 1$, la secuencia es 00, 11, 01.

Paso 1 Modelización de las especificaciones

Las especificaciones del problema, se ajustan de forma natural a un modelo Moore ya que las salidas de los elementos de memoria son también las salidas del circuito secuencial. En la Figura B.70 se muestra el diagrama de estados que describe las especificaciones del problema. Existen solo tres estados en la secuencia $S_0 = 00$, $S_1 = 01$ y $S_3 = 11$; el cuarto estado ($S_2 = 10$) podría aparecer al conectar el circuito y por esa razón se dispone que independientemente del valor de la entrada pase a un estado de la secuencia. En la Figura B.71 se muestra la tabla de estados correspondiente.

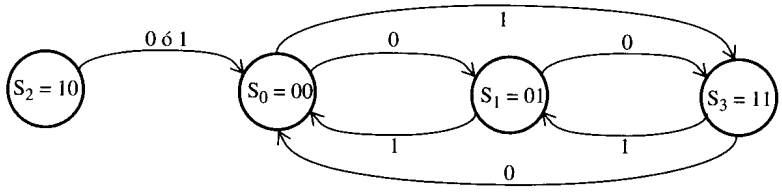


Figura B.70: Diagrama de estados del modelo de Moore del contador de dos secuencias

Estado presente	Entrada C = 0	Entrada C = 1	Salida
S_0	S_1	S_3	00
S_1	S_3	S_0	01
S_2	S_0	S_0	10
S_3	S_0	S_1	11
	Próximo estado		

Figura B.71: Tabla de estados del contador de dos secuencias

Paso 2 Minimización del número de estados

Si se aplica el método de las particiones a la tabla de estados de la Figura B.71 es fácil deducir que no es posible ninguna reducción en el número de estados.

Paso 3 Asignación de estados

En esta caso la asignación de estados viene dada de forma natural por el propio problema (ver Figura B.72).

	Q_1 Q_2
$S_0 \rightarrow$	0 0
$S_1 \rightarrow$	0 1
$S_2 \rightarrow$	1 0
$S_3 \rightarrow$	1 1

Figura B.72: Asignación de estados del contador de dos secuencias

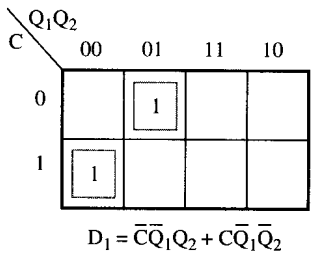
Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.73 se muestran las tablas de transición de estados que se obtienen al sustituir la asignación de estados de la Figura B.72 en la tabla de estados de la Figura B.71. En este caso no es necesario especificar la función de salida z porque coincide con el estado.

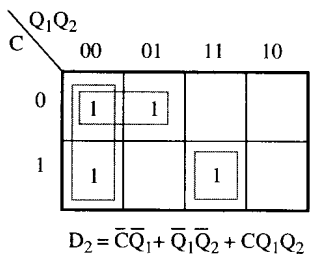
Estado presente Q_1 Q_2	Entrada C	
	0	1
0 0	0 1	1 1
0 1	1 1	0 0
1 1	0 0	0 1
1 0	0 0	0 0
	Próximo estado	

Figura B.73: Tabla de transición de estados y de salida para el contador de dos secuencias

Teniendo en cuenta la función de excitación de un elemento de memoria D ($D(t) = Q(t + 1)$)y las tablas de transición de estados de la Figura B.73, se deducen las expresiones de las entradas D_1 y D_2 a los dos elementos de memoria (ver Figura B.74).



$D_1 = C\overline{Q}_1Q_2 + C\overline{Q}_1\overline{Q}_2$



$D_2 = C\overline{Q}_1 + \overline{Q}_1\overline{Q}_2 + CQ_1Q_2$

Figura B.74: Funciones de excitación del contador de dos secuencias

Paso 5 Realización del circuito secuencial

En la Figura B.75 se muestra la realización del circuito secuencial del contador de dos secuencias.

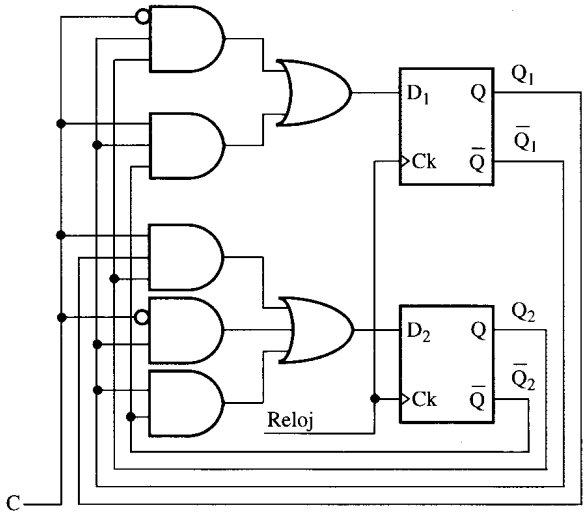


Figura B.75: Circuito secuencial del contador de dos secuencias

B.6.10 Diagramas de estado que utilizan expresiones lógicas para expresar las transiciones

Si la transición de un estado a otro estado se especifica mediante algunas variables de entrada, como podría ser el caso en un sistema digital complejo, entonces los diagramas de estado que se han construido anteriormente resultarían poco claros. Considerése por ejemplo un circuito que dispone de 3 entradas *a, b* y *c*. Supóngase que la transición del estado S_1 al estado S_2 ocurre cuando *abc* = 001 y con cualquier otra condición no ocurre ninguna transición. El diagrama de estados de la Figura B.76a muestra esta situación. Una notación más conveniente sería simplemente dar la condición booleana que es necesaria para que ocurra o no la transición tal como se muestra en la Figura B.76b. A estas expresiones lógicas se las conoce como expresiones de transición, que se pueden expresar en cualquier forma booleana que sea correcta. Por ejemplo $\overline{\bar{a}\bar{b}c}$ se podría escribir como $a + b + \bar{c}$. Si se utilizan expresiones de transición, es muy importante que todas las combinaciones de valores de las variables sean tomadas en cuenta en el diagrama. También, una combinación particular de valores solo debe hacer que sea verdadera una expresión de transición ya que en caso contrario se crearía una situación ambigua.

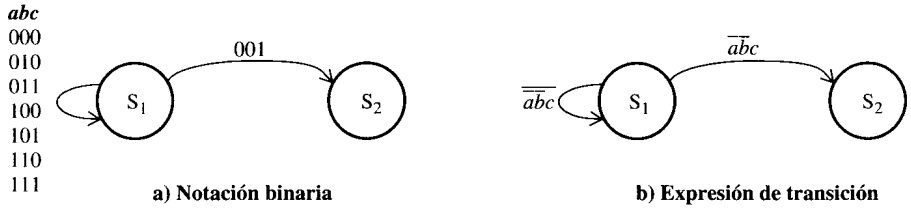


Figura B.76: Diagrama de estados con expresiones lógicas

En la Figura B.77 se muestra un diagrama de estados para un elemento de memoria tipo *JK* utilizando expresiones de transición.

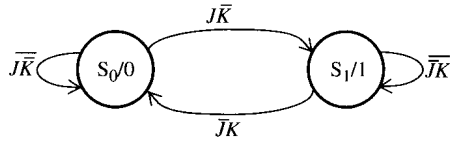


Figura B.77: Diagrama de estados del modelo de Moore de un elemento de memoria *JK* con expresiones de transición

B.6.11 Ejemplo 6: Generador de pulso con anchura de pulso variable

Especificación del problema

Diseñar un circuito secuencial síncrono con elementos de memoria tipo *D*, que posee dos entrada x_2 y x_1 y una única salida z. La salida vale 1 durante un número de ciclos de reloj que viene dado por el valor de las dos entradas. Por ejemplo, si $x_2x_1 = 01$, entonces $z = 1$durante el ciclo de reloj que sigue inmediatamente al cambio de la entrada. Si $x_2x_1 = 10$, $z = 1$ durante dos ciclos de reloj. Si $x_2x_1 = 11$, $z = 1$ durante tres ciclos de reloj. Si la entrada cambia durante el ciclo que la salida es 1, el valor de la salida retorna de forma inmediata a 0.

Paso 1 Modelización de las especificaciones

Este ejemplo que es una forma de generador de pulsos se adecua a un modelo de Moore ya que se pueden definir estados en los que la salida es 1. Para modelar las especificaciones se necesitan cuatro estados:

S_0	estado inicial
S_1	para el primer ciclo de reloj que la salida es 1
S_2	para el segundo ciclo de reloj que la salida es 1
S_3	para el tercer ciclo de reloj que la salida es 1

En la Figura B.78 se muestra el diagrama de estados utilizando expresiones de transición. Se observa que el estado S_3 retorna al estado S_0 independientemente de las condiciones de las entradas. Cada combinación posible de x_1x_2 que origina una transición particular se da como un término producto en una expresión de transición de suma de productos. Esta técnica reduce el riesgo de omitir una combinación de entrada.

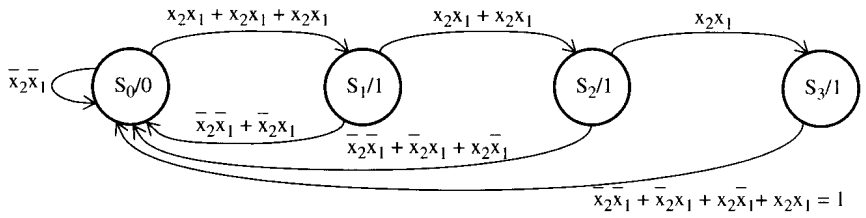


Figura B.78: Diagrama de estados con expresiones de transición

La Figura B.79 muestra la tabla de estados que se deduce del diagrama de estados de la Figura B.78.

Estado presente	Entrada (x_2 x_1) 00	01	11	10	Salida z
S_0	S_0	S_1	S_1	S_1	0
S_1	S_0	S_0	S_2	S_2	1
S_2	S_0	S_0	S_3	S_0	1
S_3	S_0	S_0	S_0	S_0	1
	Próximo estado				

Figura B.79: Tabla de estados para el generador de pulsos

Paso 2 Minimización del número de estados

Si se aplica el método de las particiones a la tabla de estados de la Figura B.79 es fácil deducir que no es posible ninguna reducción en el número de estados.

Paso 3 Asignación de estados

Se necesitan dos variables de estado para la representación de los cuatro estados del generador de pulsos. En la Figura B.80 se muestra la asignación de estados realizada. Como los estados S_1, S_2 y S_3 pueden ir al estado S_0, se les aplica la Regla 1 a dichos estados.

	Q_1	Q_2
$S_0 \rightarrow$	0	0
$S_1 \rightarrow$	0	1
$S_2 \rightarrow$	1	1
$S_3 \rightarrow$	1	0

Figura B.80: Asignación de estados para el generador de pulsos

Paso 4 Ecuaciones de entrada a los elementos de memoria

En la Figura B.81 se muestran la tabla de transición de estados y de salida que se obtiene al sustituir la asignación de estados de la Figura B.80 en la tabla de estados de la Figura B.79.

Estado presente	Entrada (x_2 x_1) 00	01	11	10	Salida z
00	00	01	01	01	0
01	00	00	11	11	1
11	00	00	10	00	1
10	00	00	00	00	1
	Próximo estado				

Figura B.81: Tabla de transición de estados y de salida para el generador de pulsos

Teniendo en cuenta la función de excitación de un elemento de memoria D ($D(t) = Q(t + 1)$) y la tabla de transición de estados y de salida de la Figura B.81, se deducen las expresiones de las entradas D_1 y D_2 a los dos elementos de memoria (ver Figura B.82).

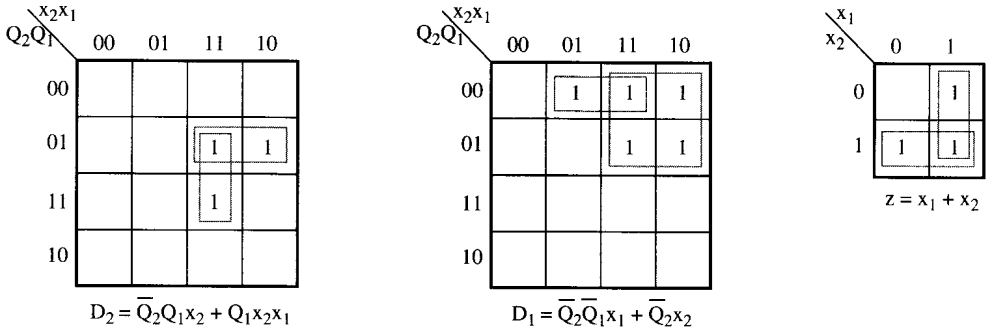


Figura B.82: Funciones de excitación y de salida del generador de pulsos

Paso 5 Realización del circuito secuencial

En la Figura B.52 se muestra la realización del circuito secuencial del generador de pulsos para la asignación de estados realizada.

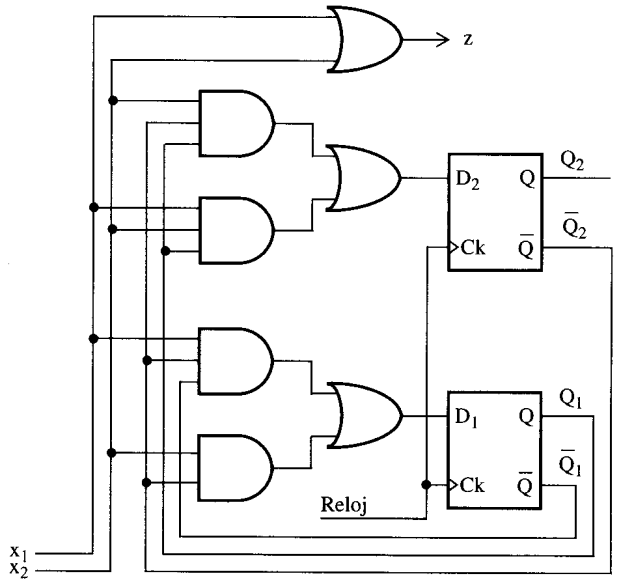


Figura B.83: Circuito secuencial del generador de pulsos

B.6.12 Tabla de estados con expresiones de transición

Cuando se utilizan expressiones de transición puede resultar útil emplear una tabla de estados que relacione

de forma explícita las expresiones de transición tal como se muestra en la tabla de la Figura B.84 para el ejemplo anterior del generador de pulsos. En esta tabla las expresiones de transición se colocan verticalmente.

Estado presente $Q_2(t)Q_1(t)$		Expresiones de transición	Próximo estado $Q_2(t+1)Q_1(t+1)$		Salida z
S_0	00	$\bar{x}_2\bar{x}_1$	S_0	00	0
		$\bar{x}_2x_1 + x_2\bar{x}_1 + x_2x_1$	S_1	01	
S_1	01	$\bar{x}_2\bar{x}_1 + \bar{x}_2x_1$	S_0	00	1
		$x_2\bar{x}_1 + x_2x_1$	S_2	11	
S_2	11	$\bar{x}_2\bar{x}_1 + \bar{x}_2x_1 + x_2\bar{x}_1$	S_0	00	1
		x_2x_1	S_3	10	
S_3	10	1	S_0	00	1

Figura B.84: Tabla de estados que muestra las expresiones de transición

Resulta entonces directo, a partir de esta tabla, escribir las ecuaciones lógicas del próximo estado sin minimizar. La forma de hacerlo es mirar las entradas para las cuales cada variable $Q_i(t+1)$ (del próximo estado $Q(t+1)$) es un 1, utilizando la siguiente ecuación:

$$Q_i(t+1) = \sum \{(\text{estados presentes que llevan a } Q_i(t+1)=1) \wedge (\text{expresión de transición})\}$$

es decir:

$$Q_2(t+1) = (\bar{Q}_2(t)Q_1(t))(x_2\bar{x}_1 + x_2x_1) + (Q_2(t)Q_1(t))(x_2x_1)$$

$$Q_1(t+1) = (\bar{Q}_2(t)\bar{Q}_1(t))(\bar{x}_2x_1 + x_2\bar{x}_1 + x_2x_1) + (\bar{Q}_2(t)Q_1(t))(x_2\bar{x}_1 + x_2x_1)$$

que son las expresiones de entrada a los elementos de memoria tipo *D*. Estas expresiones se pueden simplificar y se obtienen los mismos resultados anteriores.

Sección B-7

Síntesis de circuitos secuenciales con PLA´s o PAL´s

Los dispositivos lógicos programables (PLA´s y PAL´s) que se introdujeron en la Sección A-6 del Apéndice A se pueden utilizar para crear circuitos secuenciales. En teoría las conexiones de realimentación que posee el PLA o el PAL se pueden emplear para producir elementos de memoria. Sin embargo es preferible, tal como se muestra en la Figura B.85, incluir en el módulo elementos de memoria separados. A este circuito se le conoce como *dispositivo lógico programable con registros.*

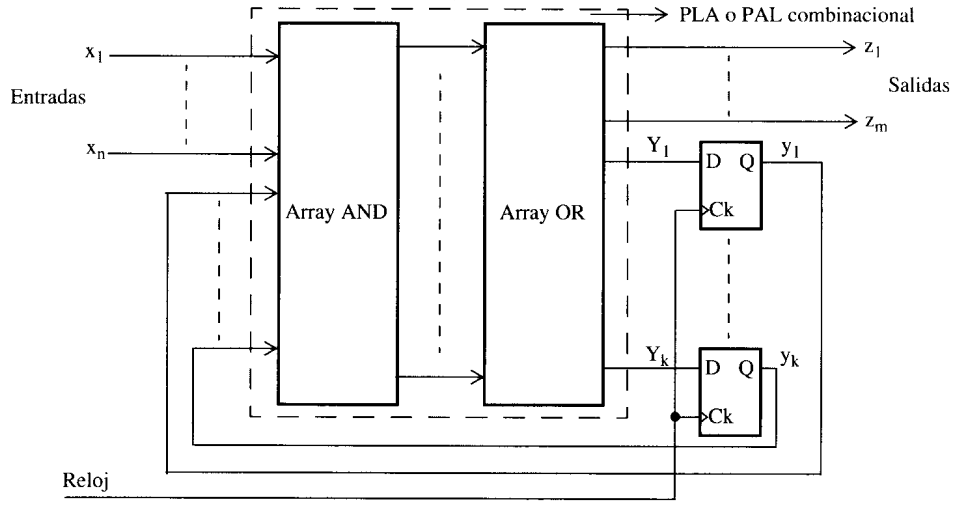


Figura B.85: Circuito secuencial síncrono que utiliza un PLA o un PAL combinacional

B.7.1 Dispositivos lógicos programables con registros

En estos módulos se integran elementos de memoria del tipo *D*, *RS* o *JK* en el propio dispositivo. El término "con registros" significa que todas las salidas se sincronizan con una única señal de reloj y por esta razón la entrada de reloj de cada elemento de memoria del módulo se conecta a una línea común de reloj.

En la Figura B.86 se muestra el diagrama del *AMD PAL16R6* que pertenece a la clase de los dispositivos lógicos programables con registros. En este módulo hay 8 entradas independientes, 6 elementos de memoria tipo *D* y 2 entradas/salidas análogas a la del *PAL16L8* de la Figura A.23. Las salidas de los elementos de memoria se pueden realimentar como variables lo que proporciona un máximo de 16 variables de la misma forma que el *PAL16L8*. Se pueden implementar un máximo de 8 términos productos en las expresiones de

sumas de términos productos. Por lo tanto para la síntesis de un circuito secuencial este módulo proporciona un máximo de 8 entradas x (x_1, x_2, ..., x_8), 6 variables de estado (Q_1, Q_2, ..., Q_6) y 2 salidas separadas (z_1, z_2) de sumas de productos.

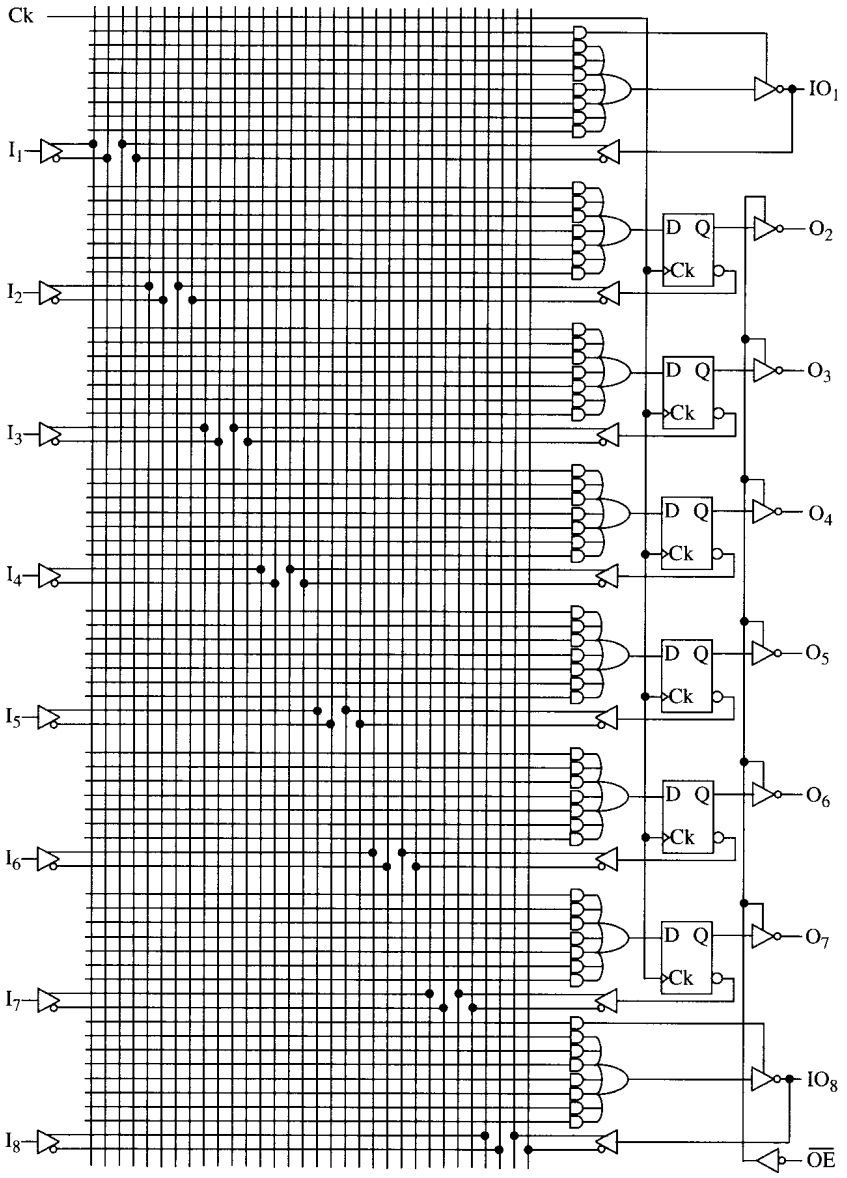


Figura B.86: Array lógico programablecon registro tipo PAL (AMD PAL16R6)

B.7.2 Ejemplo: Diseño de un contador con un PAL

Se considera otra vez el Ejemplo 5 (apartado B.6.9) que consistía en un contador síncrono que seguía una de

dos secuencias de repetición dependiendo del valor de una entrada de control C. Cuando $C = 0$, la secuencia es 00, 01, 11. Cuando $C = 1$, la secuencia es 00, 11, 01. Las ecuaciones de entrada a los dos elementos de memoria tipo D que se necesitan son:

$$D_1 = \overline{C}\,\overline{Q}_1 Q_2 + C\overline{Q}_1\,\overline{Q}_2$$

$$D_2 = \overline{C}\,\overline{Q}_1 + \overline{Q}_1\,\overline{Q}_2 + C Q_1 Q_2$$

En el módulo *PAL 16R8* se observa que a los elementos de memoria entran las funciones sin complementar mientras que sus salidas se complementan. Esto significa que si se implementan directamente las ecuaciones anteriores se obtienen de forma invertida los bits de la secuencia solicitada (es decir 11, 10, 00 ó 11, 00, 10). Si se desarrolla otra vez el Ejemplo 5 pero con los bits de la secuencia invertidos tal como se muestra en el diagrama de estados de la Figura B.87, se obtienen para las entradas a los dos elementos de memoria tipo D las siguientes expresiones :

$$D_1 = \overline{C}\,Q_2 + C Q_1\,\overline{Q}_2 + \overline{Q}_1$$

$$D_2 = \overline{C}\,\overline{Q}_1 + \overline{Q}_1 Q_2 + C Q_1\,\overline{Q}_2$$

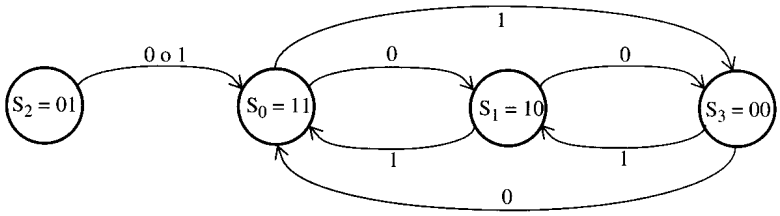


Figura B.87: Diagrama de estados del modelo de Moore del contador de dos secuencias con los bits invertidos

En la Figura B.88 se muestra la implementación de este circuito secuencial con un PAL con registros.

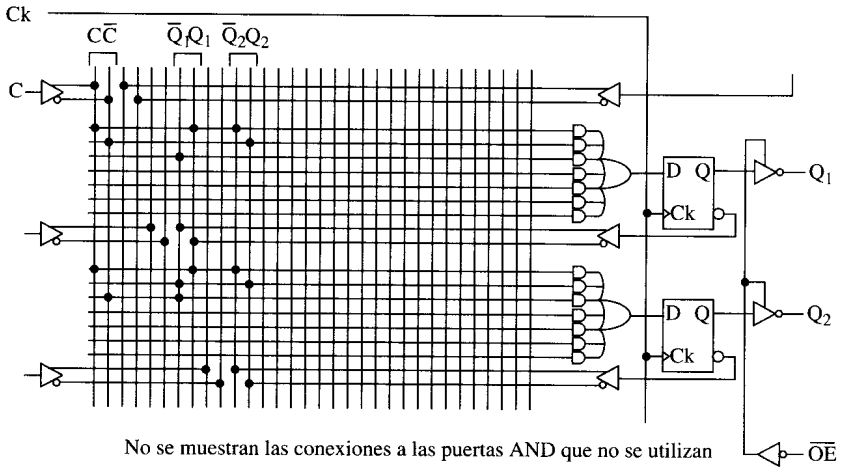


Figura B.88: Diseño del contador de dos secuencias con un PAL con registros

Existen otros módulos del tipo PAL con registros (como el *AMD PAL20X10*) que está pensado para el diseño de contadores que incluye combinaciones de puertas AND-OR-ORexclusiva tal como se muestra en la Figura B.89. Este módulo permite una implementación directa de funciones de las variables de estado que consisten en:

$$(\sum \text{términos productos}) \oplus (\sum \text{términos productos})$$

donde cada suma de términos productos puede tener dos términos de hasta 16 variables (sujetas a las limitaciones de terminales de entrada/salida compartidos, de la misma forma que en el *PAL 16R6*).

También existen PLA´s con elementos de memoria a la salida. Esto es, puertas AND-OR con conexiones seleccionables para ambos tipos de puertas y salidas con registro (elementos de memoria con un reloj común). A estos dispositivos se les denomina a veces *secuenciadores lógicos programables*.

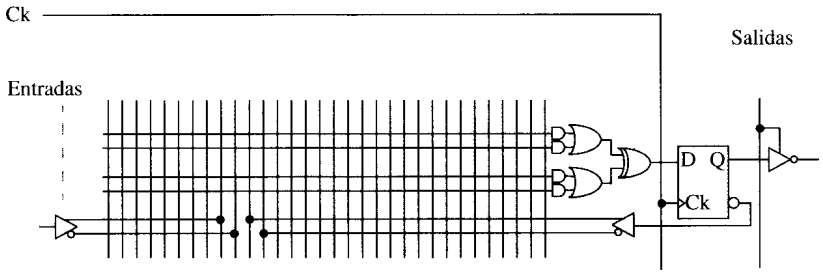


Figura B.89: Combinación de puertas AND-OR-OR exclusiva en el AMD PAL20X10

B.7.3 Macroceldas

Las *macroceldas* son circuitos que se repiten dentro de un PLA o un PAL y que poseen funciones seleccionables. En la Figura B.90 se muestra un ejemplo de una macrocelda que puede seleccionar o la salida de un elemento de memoria o una de las salidas del propio módulo. Se observa que el elemento de memoria se integra físicamente en la macrocelda.

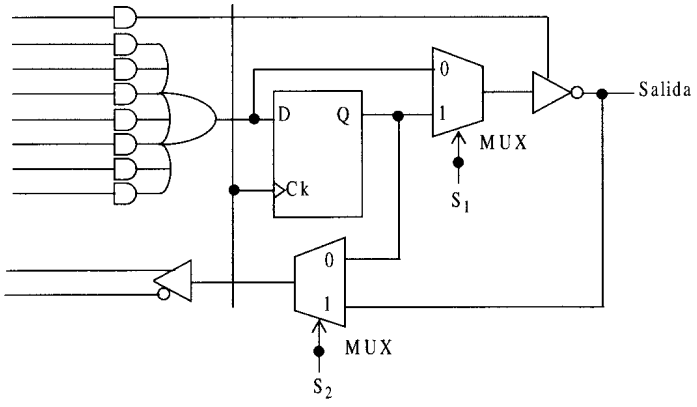


Figura B.90: Ejemplo de macrocelda con elemento de memoria incorporado

En este ejemplo, hay dos multiplexores, cada uno controlado por una conexión programable. Cuando se elimina la conexión programable, se selecciona la entrada que va etiquetada con un 1. Cuando se deja intacta la conexión programable, se selecciona la entrada que va etiquetada con un 0. En la Figura B.90 un multiplexor puede seleccionar como salida de la macrocelda o la salida Q del elemento de memoria o la función AND-OR, en ambos casos con una inversión extra a través de la puerta de salida. El otro multiplexor puede seleccionar como entrada de realimentación al array AND-OR o la salida Q del elemento de memoria o la salida del dispositivo.

Las macroceldas pueden también describir otros circuitos lógicos combinacionales/secuenciales repetidos dispuestos en arrays bidimensionales con interconexiones programables. El término macrocelda se emplea con un significado algo diferente en el diseño VLSI (Very Large Scale Integration) para identificar celdas de bibliotecas que se tienen ya prediseñadas.

Bibliografía

CERRADA, C., FELIU, V.
Estructura y Tecnología de Computadores I. Uned. Madrid (1993).

ERCEGOVAC, M. D., LANG, T.
Digital Systems and Hardware/Firmware Algorithms. John Wiley. New York (1985).

FERNÁNDEZ, G.
Conceptos Básicos de Arquitectura y Sistemas Operativos: Curso de Ordenadores (Tercera Edición). Sistemas y Servicios de Comunicación S. L. Madrid (1998).

GAJSKI, D. D.
Principios de Diseño Digital. Prentice Hall. Madrid (1997).

HAMACHER, V., VRANESIC, Z., ZAKY, S.
Computer Organization (Third Edition). McGraw-Hill. New York (1990).

HAYES, J. P.
Computer Architecture and Organization (Second Edition). McGraw-Hill. New York (1988).

HAYES, J. P.
Introduction to Digital Logic Design. Addison Wesley. Reading (1994)

HERMIDA, R., del CORRAL, A. Mª., PASTOR, E., SÁNCHEZ, F.
Fundamentos de Computadores. Editorial Síntesis. Madrid (1998).

KATZ, R. H.
Contemporary Logic Design. The Benjamin/Cummings Publishing Company Inc. Redwood (1994).

MIGUEL, P. de
Fundamentos de los Computadores. Paraninfo. Madrid (1990).

MIRA, J., DELGADO, A. E., DORMIDO, S., CANTO, Mª. A.
Electrónica Digital. Editorial Sanz y Torres. Madrid (1993).

MORILLA, F., SANCHEZ, A., VELAZQUEZ, J.
Fundamentos de los Computadores. Ceura. Madrid (1990).

MORRIS MANO, M., KIME, C. R.
Logic and Computer Design Fundamentals. Prentice Hall. New Jersey (1997).

OMONDI, A. R.
Computer Arithmetic Systems: Algorithms, Architecture and Implementations. Prentice Hall. New Jersey (1994).

PRIETO, A., LLORIS, A., TORRES, J. C.
Introducción a la Informática. Mc Graw Hill. Madrid (1995).

RAFIQUZZAMAN, M., CHANDRA, R.
Arquitectura de Ordenadores. Anaya. Madrid (1990).

SCOTT, N. R.
Computer Number Systems & Arithmetic. Prentice Hall. New Jersey (1985).

STALLINGS, W.
Organización y Arquitectura de Computadores. (Cuarta Edición). Prentice Hall. Madrid (1997).

TANENBAUM, A. S.
Organización de Computadoras un Enfoque Estructurado. (Tercera Edición). Prentice Hall. Méjico (1992).

WILKINSON, B., MAKKI, R.
Digital System Design (Second Edition). Prentice Hall. Hertfordshire (1992).

WILKINSON, B.
Computer Architecture: Design and Performance (Second Edition). Prentice Hall. Hertfordshire (1996).

ZARGHAM, M. R.
Computer Architecture: Single and Parallel Systems. Prentice Hall. New Jersey (1996).

Índice

A

Acceso directo a memoria (DMA)
configuración del DMA 168
controlador de DMA 164
DMA por ráfagas 166
DMA por robo de ciclos 166
DMA transparente 166
transferencia de datos 166
Acierto en una memoria caché 76
Algoritmo de Booth 216
Algoritmo de Booth modificado 225
Algoritmo de no restauración 233
Algoritmo de reemplazamiento en una caché
bloque elegido de forma aleatoria 90
bloque más antiguo 91
bloque utilizado menos frecuentemente 90
bloque utilizado menos recientemente 91
Algoritmo de restauración 230
Alterabilidad de una memoria
memoria de lectura-escritura (RAM) 49
memoria de sólo lectura (ROM) 49
ALU SN74x181 235
ALU´s integradas 235
Análisis de un circuito secuencial 490, 492
Anidamiento de interrupciones 155
Árbol de Wallace 222
Aritmética en coma flotante
estructura de una ALU 253
exponentes polarizados 241
Aritmética en coma flotante (*continuación*)
formato IEEE 244
multiplicación y división 250
normalización de números 239
redondeo y truncamiento 249
representación de números 239
sistemas en coma flotante 243
suma y resta 247
Arrastres en un sumador
generados 193
propagados 193
Array lógico programable (PAL) 459
Array lógico programable (PLA) 456
ASM: máquina de estados algorítmica
bloque ASM 287
bloque de condición 286
bloque de decisión 286
bloque de estado 286
diagrama 288
Asignación de estados 474

B

Base 240
BCD (decimal codificado en binario)
corrección en la suma BCD 208
restador BCD 210
sumador BCD 207
sumador BCD dígito-serie/bit-paralelo 209
sumador BCD dígito-serie/bit-serie 210
sumador BCD paralelo 209

Bit escondido 245
Bloque de una memoria caché 76
Bus
anchura 31
bidireccional 34
conexión 31
consideraciones prácticas 31
controlador 27
diagrama de temporización 21
elementos de diseño 26
estructura 22
estructura jerárquica 24
interconexión con bus de E/S 19
líneas de control 23
líneas de datos 23
líneas de dirección 23
método de arbitraje 27
temporización asíncrona 30
temporización síncrona 29
Unibus 36
unidireccional 34
arbitraje centralizado 27
arbitraje distribuido 28

C

Capacidad de una memoria 47
Características de la memoria 52
Características físicas de una memoria 49
Ciclo de escritura de una memoria 60
Ciclo de instrucción
fase de búsqueda de la instrucción 6, 350
fase de búsqueda de los operandos 353
fase de decodificación 353
fase de ejecución de la instrucción. 6, 354
Ciclo de interrupción 12, 356
Ciclo de lectura de una memoria 59
Ciclo de reloj 29
Circuito secuencial
análisis 490
asignación de estados 474
asíncrono 467
codificación binaria 474
diagrama de estados 469
estructura canónica 468
minimización del número de estados 470
modelo de Huffman-Mealy 466
Circuito secuencial (*continuación*)
modelo de Moore 467
síncrono 467
tabla de estados 469
Circuito secuencial asíncrono 467
Circuito secuencial con elementos de memoria . 490
Circuito secuencial realizado en forma canónica 490
Circuito secuencial síncrono 467
Clasificación de las interrupciones
control de la CPU 148
controlador de interrupción 158
gestión de la prioridad 149
identificación de la fuente 149
niveles 155
número de líneas 147
origen 146
Clasificación de las operaciones de desplazamiento
longitud de los registros 256
sentido del desplazamiento 255
tratamiento de los bits que rebosan 256
tratamiento del bit de signo 255
Clasificación de los PE´s
canal multiplexor 173
canal selector 173
Codificación binaria de los estados 474, 496
Codificador 441
Codificador de prioridad 154, 442
Codificador de prioridad *74147* 442
Coma flotante
aritmética 238
base 240
desbordamiento del exponente 247
desbordamiento de la mantisa 247
división 250
estructura de una ALU 253
exactitud 240
exponente 240
exponente polarizados 241
formato IEEE 244
mantisa 240
multiplicación 250
normalización de números 239
precisión 240
rango 239
redondeo 249
representación de números 239
resta 247

Coma flotante (*continuación*)
sistemas ... 243
subdesbordamiento del exponente ... 247
subdesbordamiento de la mantisa ... 247
suma ... 247
truncamiento ... 249
Comparación de números en complemento a 2 .. 265
Comparación de números positivos sin signo ... 264
Comparador combinacional ... 261
Comparador secuencial ... 263
Comparador utilizando un sumador ... 263
Complementador a 2 ... 504
Componentes de un computador
unidad aritmético- lógica (ALU) ... 3
unidad de control ... 3
unidad de entrada salida (E/S) ... 5
unidad de memoria ... 4
Componentes de un sistema digital ... 280
Conexión mediante bus ... 31
Configuración del DMA
controlador de DMA conectado al bus ... 169
bus único con DMA conectado ... 168
bus único con DMA integrado ... 169
Contador binario módulo-8 ... 505
Contador de código arbitrario ... 508
Contador módulo-*n* ... 371
Control de la CPU sobre la interrupción
enmascaramiento individual ... 148
enmascaramiento por nivel ... 148
Controlador de DMA
registro de datos ... 164
registro de dirección ... 164
registro contador de palabras ... 164
Controlador de E/S ... 131
Controlador de interrupciones ... 158
Controlador de interrupciones Intel 8259A ... 158
Controladores de interrupciones ... 158
Corrección en la suma BCD ... 208
CPU: registros
contador de programa (PC) ... 6, 350
de datos de memoria (MBR) ... 4, 351
de dirección de memoria (MAR) ... 4, 351
de instrucción (IR) ... 7, 350

D

Decodificador ... 446
Decodificador *7445* ... 448
Demultiplexor ... 454
Desbordamiento de la mantisa ... 247
Desbordamiento del exponente ... 247
Desplazamiento: operación
clasificación ... 255
estructura de los desplazamientos ... 258
registro de desplazamiento de 4 bits ... 256
Detección de la finalización del arrastre ... 199
Detector de secuencia (Huffman-Mealy) ... 497
Detector de secuencia (Moore) ... 501
Diagrama ASM ... 288
Diagrama de estados ... 469
Diagrama del ciclo de instrucción ... 10, 350, 358
Diagramas de temporización ... 21
Direccionamiento: modos
directo ... 347
implícito ... 346
indexado ... 348
indirecto ... 348
inmediato ... 346
relativo ... 347
Disco magnético
controlador del disco ... 115
estructura física ... 112
factor de entrelazado ... 117
tiempo de acceso a un archivo ... 115
planificación del disco ... 118
Disco magnético: planificación
C-LOOK ... 122
C-SCAN ... 121
FCFS ... 119
LOOK ... 122
SCAN ... 120
SSTF ... 120
Disco magnético: estructura física
cabezas de lectura escritura ... 112
cilindro ... 112
pista ... 112
Diseño de un contador con un PAL ... 516
Diseño jerárquico ... 274
Dispositivo lógico programable
array lógico programable (PAL) ... 459
array lógico programable (PLA) ... 456
Dispositivos externos ... 129
Dispositivos lógicos programables con registros 515

División
de dos números binarios sin signo 228
en coma flotante 250
método de no restauración 232
método de restauración 230
Divisor binario 228
DMA: controlador
registro de datos 164
registro de dirección 164
registro contador de palabras 164

E

E/S: estructura del sistema 134
E/S: orden
de comprobación 137
de control 137
de escritura 137
de lectura 137
E/S controlada por programa
aislada 139
bucle de espera 137
localizada en memoria 138
E/S controlada por programa en el 68000 140
E/S por interrupción 141
Ejecución de las microinstrucciones
monofásicas 400
polifásicas 400
Elemento de memoria 477
Elemento de memoria : tipos 482
D 484
JK 482
RS 482
T 484
maestro-esclavo 480
Elementos de diseño del bus 26
Enmascaramiento de interrupción 144
Especificación de un circuito secuencial 495
Estados equivalentes 472
Estrategia de escritura en una caché
escritura inmediata 92
post-escritura 93
Estructura canónica de un circuito secuencial 468
Estructura de bus 22
Estructura de interconexión
con bus de E/S 19
E/S a través de la CPU 17
Estructura de interconexión (*continuación*)
E/S a través de la memoria 24
mediante conmutador central 18
Estructura de interrupciones del 68000 162
Estructura de la unidad aritmético-lógica (ALU) 234
Estructura de los registros de desplazamiento
entrada paralelo/salida paralelo 258
entrada paralelo/salida serie 258
entrada serie/salida paralelo 258
entrada serie/salida serie 258
Estructura de un sistema digital
componentes 280
modelo de Glushkov 283
puntos de control 281
Estructura de una ALU en coma flotante 253
Estructura del controlador de E/S 133
Estructura del sistema de E/S 134
Estructura física de un disco
cabezas de lectura escritura 112
cilindro 112
pista 112
Estructura jerárquica de buses 24
Exactitud 240
Exponente 240
Exponente polarizado 241

F

Factor de entrelazado de un disco 117
Fallo en una memoria caché 76
Fases en el diseño del procesador 360
Formato de una instrucción
campo de código de operación 331
campo de dirección 331
campo de modo 332
campo de tipo de instrucción 332
Formato de una microinstrucción
campo de control 396
campo de dirección 396
campo de selección 398
codificada por subcampos 397
no codificada 396
totalmente codificada 398
Formato IEEE para números en coma flotante ... 244
Función de E/S 15
Función de un computador 6
Función lógica del registro de máscara 99

Funciones del controlador de E/S 131

G

Generador de pulso con anchura variable 511
Gestión de la prioridad de las interrupciones 149
Grado de asociatividad en una caché 87

I

Identificación de la fuente de interrupción
  mediante encadenamiento 150
  mediante encuesta 149
  mediante hardware paralelo 152
Índice de mejora λ de una caché 79
Instrucción 331
Instrucción de E/S 138
Instrucción: formato
  campo de código de operación 331
  campo de dirección 331
  campo de modo 332
  campo de tipo de instrucción 332
Instrucción: tipos
  aritmético-lógicas y de comparación 332
  de desplazamiento 332
  de gobierno 332
  de transferencia de control 332
  de transferencia de datos 332
Interconexión mediante bus
  anchura del bus 31
  bidireccional 34
  consideraciones prácticas 31
  controlador del bus 27
  diagrama de temporización 21
  estructura de bus 22
  líneas de control 23
  líneas de datos 23
  líneas de dirección 23
  método de arbitraje 27
  temporización asíncrona 30
  temporización síncrona 29
  Unibus 36
  unidireccional 34
Interfaz programable Intel 8255A 160
Interrupción
  control de la CPU 148
  controlador 158
Interrupción (*continuación*)
  enmascaramiento 144
  enmascaramiento individual 148
  enmascaramiento por nivel 148
  estructura en el 68000 162
  gestión de la prioridad 149
  identificación de la fuente 149
  mediante encadenamiento 150
  mediante encuesta 149
  mediante hardware paralelo 152
  niveles 155
  número de líneas 147
  origen 146
  petición 142
  reconocimiento 144
  registro 152

J

Jerarquía de memorias 53

L

Lógica de bifurcación 392
Lógica de selección 394
Lógica de transformación 392

M

Macroceldas 518
Mantisa 240
Máquina de decisión binaria 403
Máquina de estados algorítmica (ASM)
  bloque ASM 287
  bloque de condición 286
  bloque de decisión 286
  bloque de estado 286
Matriz de control 365
Mejoras en el algoritmo de "lápiz y papel" 214
Memoria
  acceso aleatorio 47
  acceso asociativo 49
  acceso directo 48
  acceso secuencial 48
  capacidad 47
  características 52
  características físicas 49

Memoria (*continuación*)
ciclo de escritura ... 60
ciclo de lectura ... 59
frecuencia de acceso ... 52
tiempo de acceso ... 50
tiempo de ciclo ... 51
volatilidad ... 50
Memoria asociativa
celda básica ... 98
concepto ... 96
estructura ... 96
registro de marca ... 96
registro de máscara ... 97
registro argumento ... 96
operación de lectura ... 100
operación de escritura ... 100
diseño ... 102
Memoria caché
algoritmos de reemplazamiento ... 90
capacidad ... 80
correspondencia asociativa por conjuntos ... 86
correspondencia directa ... 81
correspondencia totalmente asociativa ... 84
estrategia de escritura ... 92
grado de asociatividad ... 87
índice de mejora λ ... 79
número de cachés ... 93
organización ... 80
rendimiento ... 79
tamaño del bloque ... 93
tasa de acierto ... 79
Memoria compartida
asignación de prioridades ... 104
rotación de prioridades ... 105
Memoria de control ... 328, 387
Memoria de control
registro de datos RMC ... 388
registro de dirección RDC ... 388
Memoria de control: organización
microprogramación horizontal ... 396
microprogramación vertical ... 397
Memoria de lectura destructiva ... 50
Memoria de semiconductor
características generales ... 58
celda básica ... 60
ciclo de escritura ... 60
ciclo de lectura ... 59
Memoria de semiconductor (*continuación*)
conexión con el bus del sistema ... 71
direccionamiento ... 72
diseño de bloques de memoria ... 68
organización interna ... 62
2D ... 62
21/2D ... 65
Memoria dinámica ... 50
Memoria estática ... 50
Memoria tipo pila
aplicaciones ... 110
cumbre de la pila ... 106
extracción ... 106
inserción ... 106
realización ... 117
Método de acceso de una memoria
aleatorio ... 47
asociativo ... 49
directo ... 48
secuencial ... 48
Método de arbitraje del bus
centralizado ... 27
distribuido ... 28
Método de las particiones ... 472
Microinstrucción ... 329, 383, 387
Microinstrucción: formato
campo de control ... 396
campo de dirección ... 396
campo de selección ... 398
codificada por subcampos ... 397
no codificada ... 396
totalmente codificada ... 398
Microinstrucción: secuenciamiento
direccionamiento explícito ... 391
direccionamiento implícito ... 394
Microoperación de proceso ... 330
Microoperación de transferencia ... 330
Microoperación ... 279
Microorden ... 309
Microprograma ... 329, 383
Microprograma: representación ... 401
Minimización del número de estados ... 470, 496
Modelo de Glushkov ... 283
Modelo original de Wilkes ... 385
Modos de direccionamiento
directo ... 347
implícito ... 346

Modos de direccionamiento (*continuación*)
indexado ... 348
indirecto ... 348
inmediato ... 346
relativo ... 347
Módulo de E/S ... 16
Multiplexación en el tiempo ... 26
Multiplexor ... 451
Multiplicación
de "lápiz y papel" ... 212
en complemento a 2 ... 216
en coma flotante ... 250
Multiplicador binario
algoritmo de Booth ... 216
mejoras en el algoritmo de "lápiz y papel" ... 214
multiplicación de "lápiz y papel" ... 212
multiplicación en complemento a 2 ... 216
Multiplicador binario: diseño
diagrama ASM ... 291
ruta de datos ... 290
unidad de control ... 294
Multiplicador binario: diseño microprogramado 412
Multiplicador de alta velocidad
algoritmo de Booth modificado ... 225
árbol de Wallace ... 222
suma rápida de los productos parciales ... 222

N

Nivel de transferencia entre registros
expandibilidad de los componentes ... 278
representación ... 276
Niveles de diseño de un sistema digital
de registros ... 272
de procesador ... 273
de puertas lógicas ... 272
Niveles de interrupción ... 155
Normalización de números en coma flotante ... 239
Número de líneas de interrupción ... 147

O

Operación aritmética en coma flotante
desbordamiento del exponente ... 247
desbordamiento de la mantisa ... 247
subdesbordamiento del exponente ... 247
subdesbordamiento de la mantisa ... 247
Operación de comparación
utilizando un circuito combinacional ... 261
utilizando un circuito secuencial ... 263
utilizando un sumador ... 263
Operación de comparación ... 261
Operación de desplazamiento
clasificación ... 255
estructura de los desplazamientos ... 258
registro de desplazamiento de 4 bits ... 256
Operación de lectura (memoria asociativa) ... 100
Operación de escritura (memoria asociativa) ... 100
Orden de E/S
de comprobación ... 137
de control ... 137
de escritura ... 137
de lectura ... 137
Organización de la memoria ... 52
Organización de las memorias de semiconductor
2D ... 62
21/2D ... 65
Organización de los sumadores en código BCD
sumador dígito-serie/bit-paralelo ... 209
sumador dígito-serie/bit-serie ... 210
sumador paralelo ... 209
Organización de una memoria caché
correspondencia asociativa por conjuntos ... 86
correspondencia directa ... 81
correspondencia totalmente asociativa ... 84
Origen de las interrupciones
de programa ... 147
externas ... 146
internas ... 146

P

PAL: array lógico programable ... 459
Palabra física ... 62
Palabra lógica ... 62
Petición de interrupción ... 142
PLA: array lógico programable ... 456
PLA: síntesis de circuitos secuenciales ... 515
Planificación del disco
C-LOOK ... 122
C-SCAN ... 121
FCFS ... 119
LOOK ... 122
SCAN ... 120

Planificación del disco (*continuación*)
SSTF ... 120
Precisión flotante ... 248
Precisión ... 240
Principio de inclusión ... 94
Principio de localidad
localidad espacial ... 56
localidad temporal ... 56
Procesador CISC ... 343
Procesador con acumulador ... 336
Procesador con banco de registros ... 341
Procesador con pila ... 337
Procesador de carga/almacenamiento ... 342
Procesador de cero direcciones ... 337
Procesador de dos direcciones ... 335
Procesador de E/S (PE/S)
características ... 171
clasificación ... 172
Procesador de tres direcciones ... 333
Procesador de una dirección ... 336
Procesador RISC ... 342
Procesador SIMPLE1
diagrama ASM ... 365
diagrama de flujo ... 363
especificación ... 361
repertorio de instrucciones ... 362
unidad de control ... 366
unidad de procesamiento ... 364
Procesador SIMPLE2 ... 418
búsqueda de las instrucciones máquina ... 425
decodificación de las instrucciones máquina 425
fases en la ejecución de una μI ... 424
formato de la microinstrucción ... 423
inicialización ... 425
memoria ... 423
repertorio de instrucciones máquina ... 428
ruta de datos ... 418
unidad de control ... 421
Procesador sin ALU ... 340
Procesador: clasificación
con acumulador ... 336
con banco de registros ... 341
con pila ... 337
de carga/almacenamiento ... 342
de cero direcciones ... 337
de dos direcciones ... 335
de tres direcciones ... 333
Procesador: clasificación (*continuación*)
de una dirección ... 336
procesadores CISC ... 343
procesadores RISC ... 342
sin ALU ... 340
Puerta de palabras ... 440
Puerta triestado ... 32
Punto de control ... 281
Punto de ruptura del DMA ... 167

R

Rango ... 239
Realización cableada ... 328
Realización microprogramada ... 328
Reconocimiento de interrupción ... 144
Redondeo ... 249
Registro argumento ... 96
Registro de desplazamiento
de 4 bits ... 256
entrada paralelo/salida paralelo ... 258
entrada paralelo/salida serie ... 258
entrada serie/salida paralelo ... 258
entrada serie/salida serie ... 258
Registro de estado del 68000 ... 162
Registro de interrupción ... 152
Registro de marca ... 96
Registro de máscara ... 97
Registro de máscara de interrupción ... 148, 153
Registros de la CPU
contador de programa (PC) ... 6, 350
de datos de memoria (MBR) ... 4, 351
de dirección de memoria (MAR) ... 4, 351
de instrucción (IR) ... 7, 350
Reglas de asignación de estados ... 475
Reloj ... 29, 467, 480
Reloj bifásico ... 487
Repertorio de instrucciones ... 331
Representación de números en coma flotante
base ... 240
exactitud ... 240
exponente ... 240
mantisa ... 240
precisión ... 240
rango ... 239
Representación de puntos de un sumador ... 183
Representación en el formato IEEE ... 245

Resta en coma flotante 247
Restador en código BCD 210

S

SCSI (Small Computer System Interface) 25
Secuenciador 388
Secuenciamiento explícito 333
Secuenciamiento implícito 333
Semisumador binario (SSB)
arrastre de un SSB 184
realización lógica de un SSB 185
Síntesis de circuitos secuenciales con PLA 515
Síntesis de un circuito secuencial 495
Sistema con dos niveles de memoria 55
Sistema digital: niveles de diseño
de registros 272
de procesador 273
de puertas lógicas 272
Sistemas en coma flotante 243
Subdesbordamiento de la mantisa 247
Subdesbordamiento del exponente 247
Suma rápida de los productos parciales 222
Suma 247
Sumador binario
semisumador binario (SSB) 183
sumador binario completo (SBC) 184
sumador binario paralelo SPA 188
sumador binario serie 187
sumador-restador binario paralelo 189
Sumador binario 182
Sumador binario completo (SBC)
arrastre de un SBC 186
realización lógica de un SBC 187
Sumador binario paralelo
con propagación del arrastre 188
Sumador binario serie 187
Sumador con aceleración de arrastres CAA 195
Sumador con anticipación del arrastre
SBC modificado 194
CAA circuito de aceleración de arrastres 195
Sumador con arrastre almacenado CSA 203
Sumador con selección del arrastre 199
Sumador de alta velocidad
características de los arrastres 193
con anticipación del arrastre 193
con arrastre almacenado 203
Sumador de alta velocidad (continuación)
con selección del arrastre 199
de suma condicional 197
detección de la finalización del arrastre 199
que minimiza el número de arrastres 202
Sumador de suma condicional 197
Sumador en código BCD 207
Sumador que minimiza el número de arrastres 202
Sumador-restador binario paralelo
con propagación del arrastre 189
detección del rebose 190

T

Tabla de estados 469
Tabla de estados con expresiones de transición 513
Tasa de acierto *h* de una caché 79
Tipo físico de memoria
mágnética 49
magneto-óptica 49
óptica 49
semiconductor 49
Transferencia a un subprograma 355
Transferencia de E/S en el 68000 140
Transferencia del disco a la memoria
retardo rotacional 114
tiempo de acceso 114
tiempo de búsqueda 114
tiempo de transferencia 114
Transición condicional 402
Transición incondicional 402
Truncamiento 249

U

Unibus 36
Unidad de control del multiplicador
con elementos de memoria tipo D 297
con lógica cableada 294
con un contador y un decodificador 312
con un elemento de memoria por estado 304
con un registro y un decodificador 300
con un registro y un PLA 315
con un registro y una memoria ROM 307
Unidad de control microprogramada
memoria de control 387
secuenciador 388

Unidad de control 327
Unidad de procesamiento 327
Unidad de transferencia de una memoria 47

V

Velocidad de una memoria
frecuencia de acceso 52
Velocidad de una memoria (*continuación*)
tiempo de acceso 50
tiempo de ciclo 51
Volatilidad de una memoria 50

W

Wilkes 385