# Fundamentos de computadores

PID_00160302

Material docente de la UOC

www.uoc.edu

**Montse Peiron Guàrdia**

Doctora en Informática (1996). Profesora del Departamento de Arquitectura de Computadores de la Universidad Politécnica de Cataluña (1990-2000).

**Lluís Ribas i Xirgo**

Licenciado en Informática (1989) y doctorado en Informática (1996) por la Universidad Autónoma de Barcelona. Profesor titular del Departamento de Microelectrónica y Sistemas Electrónicos desde 1989 de la misma universidad. Consultor de la Universitat Oberta de Catalunya desde 1997. Actualmente, imparte docencia de asignaturas tanto de software como de hardware en la Escuela de Ingeniería (EE) de la UAB. También ha impartido o imparte docencia en cursos de doctorado y de máster, especialmente en relación con sistemas de agentes físicos con controladores hardware/software empotrados y distribuidos. En la vertiente investigadora es jefe del grupo de investigación SHADES.

**Fermín Sánchez Carracedo**

Doctor en Informática desde 1996 por la Universidad Politécnica de Cataluña (UPC). Desde 1987, profesor titular de dicha universidad, adscrito al Departamento de Arquitectura de Computadores, y desde 1997, profesor consultor de la Universitat Oberta de Catalunya (UOC). Sus áreas de investigación principales son el desarrollo de nuevas arquitecturas multihilo para procesadores VLIW, la sostenibilidad en las tecnologías de la información y el desarrollo de nuevas estrategias educativas para adaptar los estudios universitarios actuales al Espacio Europeo de Educación Superior.

**A. Josep Velasco González**

Miembro de la Unidad de Microelectrónica del Departamento de Informática de la Universidad Autónoma de Barcelona desde 1989. Doctorado en Informática en 1997 en la especialidad de Microelectrónica. Desarrolla la actividad docente y de investigación en el campo del diseño e implementación de los circuitos digitales en la Escuela de Ingeniería de la UAB.

**Ramon Costa Castelló**

Doctor en Informática por la Universidad Politécnica de Cataluña (UPC). Desde 1993 trabaja en el Instituto de Organización y Control de Sistemas Industriales (IOC) y en el Departamento de Ingeniería de Sistemas, Automática e Informática Industrial (ESAII) de la UPC, donde realiza diferentes tareas de investigación y docencia en el ámbito de los sistemas informáticos de tiempo real y el control por computador.

El encargo y la creación de este material docente han sido coordinados por los profesores: Montse Serra Vizern, David Bañeres Besora

Primera edición: septiembre 2011
© Montse Peiron Guàrdia, Lluís Ribas Xirgo, Fermín Sánchez Carracedo, A. Josep Velasco González
Todos los derechos reservados
© de esta edición, FUOC, 2011
Av. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Material realizado por: Eureca Media, SL
Depósito legal: B-27.712-2011
ISBN: 978-84-693-9186-0

*Los textos e imágenes publicados en esta obra están sujetos -excepto que se indique lo contrario- a una licencia de Reconocimiento-Compartir igual (BY-SA) v.3.0 España de Creative Commons. Se puede modificar la obra, reproducirla, distribuirla o comunicarla públicamente siempre que se cite el autor y la fuente (FUOC. Fundació per a la Universitat Oberta de Catalunya), y siempre que la obra derivada quede sujeta a la misma licencia que el material original. La licencia completa se puede consultar en: http://creativecommons.org/licenses/by-sa/3.0/es/legalcode.ca*

# Contenidos

Módulo didáctico 1
**Introducción a los fundamentos de los computadores**
A. Josep Velasco González
1. El estudio de los fundamentos de los computadores
2. La evolución de los computadores
3. ¿Cómo son los computadores digitales actuales?

Módulo didáctico 2
**Representación de la información**
A. Josep Velasco González
1. Los números y los sistemas de representación
2. Representación de los números en un computador
3. Otros tipos de representaciones

Módulo didáctico 3
**Los circuitos lógicos combinacionales**
Montse Peiron Guàrdia, Fermín Sánchez Carracedo
1. Fundamentos de la electrónica digital
2. Implementación de circuitos lógicos combinacionales
3. Bloques combinacionales

Módulo didáctico 4
**Los circuitos lógicos secuenciales**
Montse Peiron Guàrdia, Fermín Sánchez Carracedo
1. Caracterización de los circuitos lógicos secuenciales
2. El biestable D
3. Bloques secuenciales
4. El modelo de Moore

Módulo didáctico 5
**Estructura básica de un computador**
Lluís Ribas i Xirgo
1. Màquinas de estados
2. Máquinas algorítmicas
3. Arquitectura básica de un computador

# Introducción a los fundamentos de los computadores

A. Josep Velasco González

PID_00163597

Universitat Oberta de Catalunya
www.uoc.edu

# Índice

**Introducción** .................................................. 5

**Objetivos** .................................................. 6

**1. El estudio de los fundamentos de los computadores** .............. 7
1.1. ¿Por qué estudiar los fundamentos de los computadores? .......... 7
1.2. ¿Qué tenemos que saber para entender los computadores? ........ 7
1.2.1. ¿Qué es la electrónica digital? .................................. 8
1.2.2. La codificación de la información .............................. 9
1.2.3. Los sistemas digitales .............................................. 10

**2. La evolución de los computadores** .................................. 12
2.1. Primera generación (1940-1955) .......................................... 13
2.2. Segunda generación (1955-1965) ......................................... 14
2.3. Tercera generación (1965-1970) ........................................... 15
2.4. Cuarta generación (1970-) .................................................. 15

**3. ¿Cómo són los computadores digitales actuales?** .................. 17
3.1. Arquitectura de Von Neuman ............................................... 18
3.2. La arquitectura de Harvard ................................................ 20

**Resumen** .................................................. 21

**Bibliografía** .................................................. 23

# Introducción

Actualmente, el uso de los ordenadores está plenamente generalizado en nuestra sociedad. Los encontramos por todas partes. Se han convertido en una herramienta de la que conocemos un buen número de funcionalidades y a la que damos un montón de aplicaciones muy variadas para facilitar nuestro trabajo o mejorar nuestra calidad de vida.

El éxito de los computadores digitales deriva del hecho de que son máquinas de propósito general, que pueden ser programadas para casi cualquier tarea si se dispone de la interfaz adecuada. Puede ser, al mismo tiempo, una herramienta de trabajo y un dispositivo de ocio. La misma máquina puede llevar a cabo cálculos sofisticados para hacer simulaciones de procesos, convertirse en una herramienta de precisión para dibujar planos o gestionar con eficiencia una base de datos compleja.

En este módulo presentamos la estructura básica de los computadores actuales, la evolución que han sufrido hasta llegar al estado actual y una introducción a los conceptos que se irán desarrollando a lo largo de los módulos siguientes, para entender en profundidad el funcionamiento de los computadores digitales.

# Objetivos

El objetivo de este módulo es presentar la estructura básica de un computador digital, poniendo de manifiesto los conocimientos que se trabajarán para entender en profundidad su funcionamiento y su diseño. Con este módulo se persigue:

1. Saber diferenciar entre la electrónica digital y la electrónica analógica.

2. Entender que es posible codificar cualquier información con un conjunto reducido de símbolos, como 0 y 1.

3. Conocer a grandes rasgos, la evolución de los computadores y las mejoras tecnológicas que han marcado cambios cualitativos profundos.

4. Conocer la arquitectura básica de un computador digital actual.

Entender la estructura básica de un computador digital es el objetivo final del curso. En este módulo se describe la arquitectura de un computador a grandes rasgos. A lo largo de los módulos siguientes se irán presentando conceptos, herramientas y metodologías para entender con profundidad el funcionamiento y la construcción de este tipo de máquinas.

# 1. El estudio de los fundamentos de los computadores

## 1.1. ¿Por qué estudiar los fundamentos de los computadores?

Desde el principio nos podríamos cuestionar la utilidad de analizar el funcionamiento de los computadores. El argumento para ello puede ser que queremos utilizar los computadores sólo como una herramienta, que en último término seremos usuarios de las máquinas y que, como tales, el conocimiento de la organización interna del computador tiene poca utilidad. La conclusión sería que se trata de una materia que tiene interés para un número reducido de ingenieros, sólo para aquellos que tienen en su horizonte trabajar en el desarrollo de procesadores.

Sin embargo, el conocimiento de los principios de funcionamiento de los computadores es necesario tanto si nos dedicamos al desarrollo de aplicaciones, al análisis de sistemas o al desarrollo de circuitería específica. El desarrollo de aplicaciones optimizadas, requiere del conocimiento de los paradigmas básicos de funcionamiento de las máquinas donde se ejecutarán, y éstas se extienden en un abanico de aplicaciones que va desde los PLC industriales a la inteligencia artificial.

**PLC**

PLC es la sigla de *programmable logic controller* y se trata de un equipamiento electrónico programable diseñado para controlar procesos secuenciales en un entorno industrial.

Los computadores son sistemas digitales complejos. Entenderlos y conocer herramientas metodológicas para su diseño y síntesis nos abre el camino al desarrollo de sistemas digitales específicos.

No se trata sólo de conocimientos de cultura general. Los conceptos básicos del funcionamiento de los computadores son conocimientos necesarios para aquel que quiera que trabajar en el diseño de sistemas electrónicos, en la programación de los mismos o en el desarrollo de aplicaciones específicas que requieran un cierto grado de optimización.

## 1.2. ¿Qué tenemos que saber para entender los computadores?

Los computadores actuales son aparatos electrónicos. La electrónica, finalmente, trabaja con señales eléctricas. ¿Cómo podemos procesar la información del mundo que nos rodea mediante señales eléctricas?

Hemos de saber cómo se codifica la información que tenemos que procesar dentro de las máquinas. Tenemos que determinar cómo son los datos y cuáles son las limitaciones implícitas en las máquinas. La matemática nos da herramientas para codificar adecuadamente la información que queremos almacenar o con la que queremos trabajar dentro de los computadores.

Los computadores se basan en la electrónica digital. Sin embargo, ¿qué es la electrónica digital? ¿En qué se diferencia de la que no es digital? En definitiva, ¿cuáles son las bases de funcionamiento de la tecnología con la que se diseñan las máquinas digitales?

Por otra parte, ¿cómo podemos utilizar la electrónica digital para construir un computador digital? Tenemos que disponer de metodologías que, de manera organizada, nos permitan concebir sistemas digitales complejos, y en particular, concretarlo sobre la organización de un computador convencional.

Éstos son los interrogantes a los que iremos dando respuesta a lo largo del curso, pero, a modo de introducción, los apartados siguientes nos dan algunas pinceladas al respecto.

### 1.2.1. ¿Qué es la electrónica digital?

Se llama **electrónica digital o discreta** a la electrónica basada en señales sobre las que sólo se identifica un conjunto finito de valores diferentes (habitualmente dos).

En contraposición, en la electrónica analógica las señales pueden variar de forma continua, es decir, no están reducidas a un conjunto (pequeño) de valores diferentes. En una señal digital sólo se diferencia entre el valor alto de tensión y el valor bajo de tensión, por ejemplo 0 V y 5 V. En cambio, una señal analógica puede registrar cualquier valor de tensión, 0,1 V o 0,2 V o 0,23 V o 2,35 V o 1,13 V o cualquier otro dentro de los márgenes de funcionamiento, y cada valor se considera diferente.

**v es el símbolo que identifica la unidad de medida del voltaje, el voltio.**

Las tecnologías actuales con las que se construyen los sistemas digitales (es decir, los dispositivos basados en la electrónica digital y los computadores en particular) trabajan especialmente bien cuando sobre las señales tan sólo se identifican dos valores de tensión diferentes. Estos valores reciben denominaciones diferentes según el ámbito de trabajo, como **verdad y falso** o bien **0 y 1 lógicos**.

Esto quiere decir que, como sistema digital, toda la información que deba procesar un computador tiene que estar codificada de forma adecuada, utilizando sólo los dos valores de tensión posibles, lo que llamamos 0 y 1 lógicos.

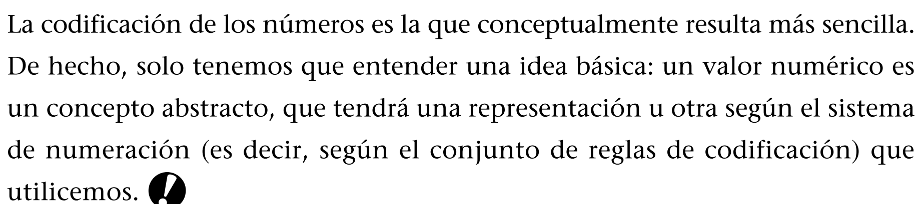
Vivimos en un mundo analógico y nos parece natural registrar la información de manera analógica. Sin embargo, trabajar directamente con información analógica resulta poco práctico y nada adecuado si queremos procesar esta in-

formación en un computador digital. Habrá que disponer de mecanismos para digitalizar la información, es decir, para codificarla utilizando sólo ceros y unos.

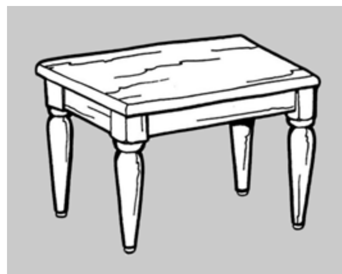
### 1.2.2. La codificación de la información

Quizás no hayamos caído en la cuenta de que, en realidad, toda la información está siempre codificada de una manera u otra. Cuando escribimos, codificamos la información en palabras que pueden estar compuestas por un conjunto de símbolos diferentes (las letras del abecedario). Cualquier valor numérico lo codificamos mediante un conjunto de símbolos que llamamos **dígitos**. Pues bien, los computadores digitales actuales gestionan información codificada utilizando los valores 0 y 1.

La codificación de los números es la que conceptualmente resulta más sencilla. De hecho, solo tenemos que entender una idea básica: un valor numérico es un concepto abstracto, que tendrá una representación u otra según el sistema de numeración (es decir, según el conjunto de reglas de codificación) que utilicemos.



Dicho de esta forma, puede parecer un poco extraño, pero estamos muy acostumbrados a esta idea. Observemos la imagen siguiente:



Unos pensaremos *mesa,* otros *taula*, otros *table*, etc. La imagen es la misma para todos, pero es posible que la tengamos asociada a palabras distintas, de hecho, con letras distintas y, si nuestra lengua es el árabe o el chino, con signos distintos. Por lo tanto, estamos codificando esta información según nuestro sistema de representación.

Con los números pasa exactamente lo mismo. Un determinado valor numérico es independiente del sistema de representación que utilicemos. Para poder codificar los valores numéricos sólo con ceros y unos, tenemos que utilizar un sistema de numeración adecuado, diferente al sistema decimal al que estamos acostumbrados.

Todo parece indicar que, muy al inicio, para referirse por ejemplo a un conjunto de cinco ovejas, el hombre dibujaba literalmente cinco ovejas. Después consiguió separar el valor numérico del objeto, por ejemplo, dibujando una

única oveja y cinco rayas o puntos o marcas de cualquier tipo. Con toda probabilidad aprendió a dar nombre a este valor numérico independiente del objeto al que se aplicaba.

No debía de ser nada práctico tener un nombre para cada valor numérico (demasiados nombres a recordar), así que se empezaron a hacer grupos para facilitar los recuentos de conjuntos “grandes”. Claro está que la cantidad de elementos de un grupo tenía que ser fácil de recordar, especialmente cuando el sistema se extendió para trabajar con grupos de grupos. En este asunto, la anatomía humana ha tenido bastante que ver y, por este motivo, los “grupos” que más se adoptaron fueron los de cinco, los de diez y los de veinte, coincidiendo con el número de dedos de una mano, de dos manos o de manos y pies.

De entre éstas, la base 10 ha salido ganadora (quizás por la aparición del calzado, quién sabe) y la idea de grupos de grupos acabó desembocando en un sistema de numeración posicional como el que tenemos ahora, donde la posición que ocupa un dígito está asociada a un grupo de grupos (decimos un peso), lo que facilitó enormemente el desarrollo de la aritmética.

Pues bien, dentro de los computadores tenemos que adaptar el sistema de numeración a su propia “anatomía”. Trabajan utilizando señales sobre las que diferencian dos niveles de tensión. Por lo tanto, tendremos que utilizar un sistema de numeración en base 2. Además, cambiar el sistema de numeración conlleva cambios en la manera de calcular el resultado de las operaciones aritméticas. Es decir, el concepto de suma es independiente del sistema de numeración, pero la forma de hacer la suma depende de la forma como representemos los números.

Todas estas cuestiones se tratan en el segundo módulo, donde se analiza nuestro sistema de numeración y se adapta a las características de las máquinas, además de identificar las limitaciones propias de las máquinas.

### 1.2.3. Los sistemas digitales

Hemos hecho una introducción al concepto de electrónica digital. Habrá que ver, sin embargo, qué es lo que la hace atractiva, adecuada para el procesamiento de información, cuáles son las herramientas que nos permiten construir circuitos complejos para el procesamiento de información y, en último término, computadores digitales de propósito general.

Conceptualmente, la electrónica digital es la electrónica de los números. Aquí, las señales eléctricas representan números. Son fáciles de codificar y resistentes a la degradación con una codificación adecuada. En los sistemas analógicos, que trabajan con ondas, la información está contenida en la forma de la onda, que se puede degradar fácilmente y que, por lo tanto, es susceptible de perder

información con facilidad, además de requerir circuitería específica para cada aplicación.

Intentar construir o entender el funcionamiento de circuitos digitales complejos, como los computadores, es una tarea inviable si no se dispone de las herramientas y de las metodologías que permitan sistematizar, en cierta medida, la construcción de sistemas digitales complejos. En este sentido, se establece una diferenciación importante entre los circuitos digitales combinacionales y los circuitos digitales secuenciales, es decir, entre los circuitos con capacidad de memoria (los segundos) y los que no la tienen (los primeros).

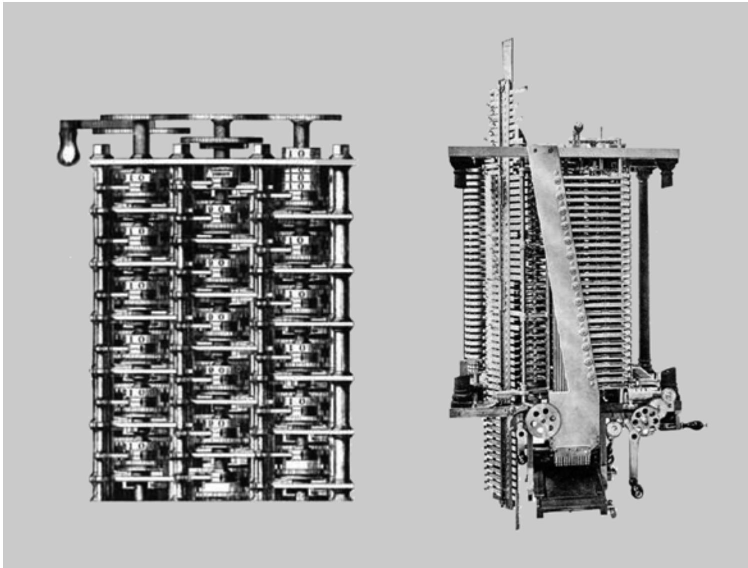
El módulo 3, dedicado a los circuitos combinacionales, y el módulo 4, donde se trabajan los circuitos secuenciales, se encargan de hacer una introducción a los sistemas digitales y a las herramientas que nos ayudan en su concepción y análisis.

Los dos apartados siguientes de este módulo introductorio están dedicados al computador digital. En el primero encontraréis una descripción del camino que se ha seguido desde los primeros ingenios de cálculo hasta los computadores actuales. Se describen características y técnicas que han ido apareciendo a lo largo de los años y que se acumulan en los ordenadores actuales. En el segundo apartado se muestra la arquitectura básica de los computadores actuales. El módulo 5 está dedicado a un análisis de la arquitectura básica que aquí se describe.

## 2. La evolución de los computadores

Desde hace siglos, se ha perseguido una mejora en el procesamiento de información, especialmente, en cálculos aritméticos, para lo cual se ha utilizado la tecnología existente en cada momento. Los primeros intentos dieron lugar a toda una serie de ingenios mecánicos, básicos como el ábaco, o realmente elaborados y complejos como la máquina diferencial de Charles Babbage.

Máquina analítica de Charles Babbage



Fuente: Bettman Archive

**Charles Babbage** (1791-1871) ocupa un lugar especialmente destacado en la historia de la computación por la concepción de la **máquina analítica** que incorpora por primera vez el concepto de máquina dirigida por un programa externo. El diseño de la máquina analítica incluía una memoria (mecánica), una unidad de procesamiento, una unidad de control (constituida por "barriles" similares a los cilindros de las cajas de música), una entrada de datos (inspirada en las tarjetas perforadas del telar de Jacquard) y salida por impresión (similar a la máquina de escribir).

El descubrimiento de la energía eléctrica permitió el desarrollo de máquinas electromecánicas que incluían lectores de tarjetas y procesamiento con conmutadores. De entre este tipo de máquinas destaca la **máquina tabuladora** de **Herman Hollerith** (1860-1929), que fue escogida para ayudar en el censo de los Estados Unidos en el año 1890. El censo manual tardaba cerca de 10 años, pero con la máquina tabuladora, que leía y procesaba (básicamente contaba) las tarjetas perforadas diseñadas al efecto, el tiempo se redujo a menos de 3 años. Herman Hollerith es considerado el primer informático, el primero en hacer un tratamiento automatizado de la información.

Máquina tabuladora de Herman Hollerith

Creative CommonsAttribution 2.0 Generic
Fuente: http://en.wikipedia.org

Las máquinas electromecánicas llegaron a convertirse en los primeros computadores digitales. **Konrad Zuse** (1910-1995) concibió la **Z1**, que disponía de memoria mecánica binaria, la **Z2**, que realizaba el procesamiento a partir de relés y mejoras que se convirtieron en las **Z3** y **Z4**. **George Stibitz** (1904-1995) concibió computadores de relés para los laboratorios Bell, y **Howard Aiken** (1900-1973) es el responsable de la serie **Mark** para la Universidad de Harvard. Éstas fueron las primeras máquinas desarrolladas con propósito comercial.

La revolución electrónica en la computación se inicia durante la Segunda Guerra Mundial. El conflicto bélico había animado el desarrollo de dispositivos electrónicos, y las experiencias en máquinas electromecánicas hicieron que enseguida se viera la aplicación de estos dispositivos a la computación.

La era de los computadores electrónicos se divide en cuatro generaciones atendiendo a los progresos en la tecnología. Los saltos generacionales vienen determinados por cambios tecnológicos. Dentro de cada generación aparecen diferentes técnicas o conceptos que se han convertido en esenciales para los computadores actuales.

### 2.1. Primera generación (1940-1955)

Esta primera generación está marcada por el uso de válvulas de vacío y la introducción de la tecnología de anillos de ferrita para la memoria. Son computadores de esta primera generación:

- **ENIAC.** J. Mauchly; J. P. Eckert (1941-1945). *Electronic Numerical Integrator And Computer.* Moore School of Engineering (Pennsylvania U.). Este ingenio constaba de 18.000 válvulas de vacío, 70.000 resistencias y 10.000 condensadores. Ocupaba un espacio de 100 m$^2$, pesaba 30 tm y tenía un consumo de 140 kw/h.

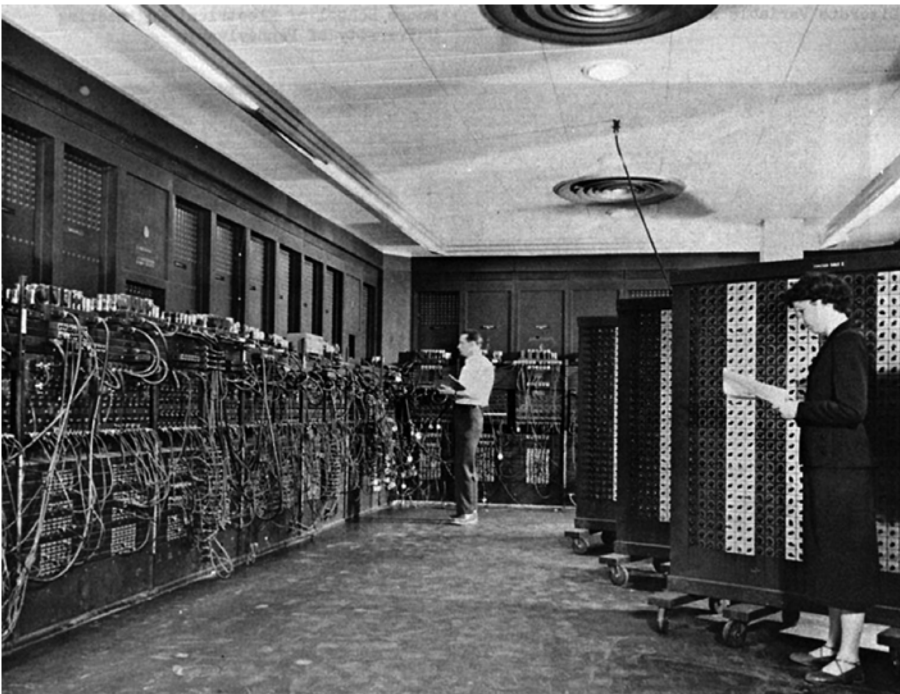
**Consumo ENIAC**

Para valorar el consumo eléctrico del computador ENIAC (140 kw/h) lo podemos comparar con un electrodoméstico de consumo elevado: el consumo de un horno eléctrico está en torno a los 2 kw/h.

- **EDVAC** (1952!). *Electronic Discrete Variable Automatic Computer*, de dimensiones más reducidas que el ENIAC. Es especial porque es la máquina sobre la que J. Von Neumann en 1945 escribió su *First Draw of a Report on the EDVAC*, en la Moore School, el primer documento donde se describe el concepto de **programa almacenado**, que forma parte de la base de los computadores actuales. También es de destacar el uso, por primera vez, de la **aritmética binaria**, en detrimento de la decimal.

- **UNIVAC** (1951). *Universal Automatic Computer*. Ecker-Mauchly Company. Con 5.400 válvulas y 1.000 palabras de memoria presenta la característica de **programa parcialmente almacenado**.

Programación del ENIAC. Imagen bajo dominio público



Fuente: http://es.wikipedia.org

## 2.2. Segunda generación (1955-1965)

El paso a la segunda generación viene marcado por la utilización de los transistores en sustitución de las válvulas de vacío. Son máquinas de esta generación:

- **PDP-1** de DEC, aparecida en 1960, que presenta por primera vez un **terminal gráfico**. Sobre esta máquina corrió el primer videojuego.

- **IBM 7030**. Esta máquina de 1961 incorpora la idea de **segmentación de memoria** y **de memoria virtual**, técnicas con las que se consiguió mejorar sensiblemente la capacidad, la gestión y el rendimiento de la memoria.

- **ATLAS** de Ferranti Ltd. & U. Manchester, 1962. Se trata de uno de los primeros **supercomputadores.** Tecnológicamente destaca por la incorporación del uso de lo que se denominan **interrupciones** para controlar los periféricos.

- **CDC 6600**. S. Cray. Control Fecha Corp., 1964. Con una velocidad de cálculo de 1 megaFLOPS (un millón de operaciones de coma flotante por segundo) conseguida gracias al paralelismo de las unidades de cálculo, ostentó el título de máquina más rápida entre 1964 y 1969.

### 2.3. Tercera generación (1965-1970)

La aparición de los primeros circuitos integrados marca el final de la segunda generación de computadores y el inicio de la tercera. Los circuitos integrados aportan una reducción de espacio significativa, una reducción importante del consumo y un aumento de la fiabilidad, que da lugar a la aparición de los primeros **minicomputadores**. De esta generación podemos destacar:

- **IBM 360**, 1964. Inicia la primera serie de computadores compatibles (seis en total), es decir, que podían utilizar el mismo software y los mismos periféricos.

- **DEC PDP/8**, 1965. Primer minicomputador de éxito comercial. Como innovaciones presentaba circuitos lógicos en módulos integrados (*chips*) y un conjunto de líneas de conexión en paralelo para interconectar los módulos: **el bus**.

- **IBM 360/85**, 1968. Es la primera en incorporar el concepto de **memoria *caché***, técnica que reduce enormemente el tiempo de acceso a la memoria y que se ha convertido en un elemento central de los sistemas actuales.

### 2.4. Cuarta generación (1970-)

Las mejoras en el proceso de fabricación de circuitos integrados conducen a un aumento considerable de la densidad de integración. Es este aumento en la densidad de integración lo que permite integrar todos los circuitos de la unidad central de proceso en un único *chip*: nacen los **microprocesadores**, el primero de los cuales es el Intel 4004 en 1971.

La cuarta generación se inicia con el desarrollo de este microprocesador. Al mismo tiempo, y debido también a las mejoras en los procesos de fabricación de circuitos integrados, se abandonan las memorias de ferritas y se incorporan las **memorias de semiconductores**. El campo de los computadores personales está sembrado, y pronto germina:

- **Altair 8800**, 1975. Se considera el primer computador personal.

- **Supercomputador Cray 1**, 1976. Incorpora por primera vez el procesamiento paralelo.

- **IBM PC**, 1981. Con el microprocesador Intel 8086 y el sistema operativo Microsoft DOS marca el inicio de la revolución de la computación personal.

- **Lisa (Apple**), 1983. Incorpora un nuevo dispositivo revolucionario, el ratón y una interfaz de usuario gráfico (estilo Windows).

IBM PC

Creative CommonsGenérica de Atribución/Compartir-Igual 3.0
Fuente: http://es.wikipedia.org

# 3. ¿Cómo són los computadores digitales actuales?

En términos generales, un **computador** es un dispositivo construido con el propósito de manipular o transformar información para conseguir una información más elaborada, como por ejemplo, el resultado de un problema determinado.

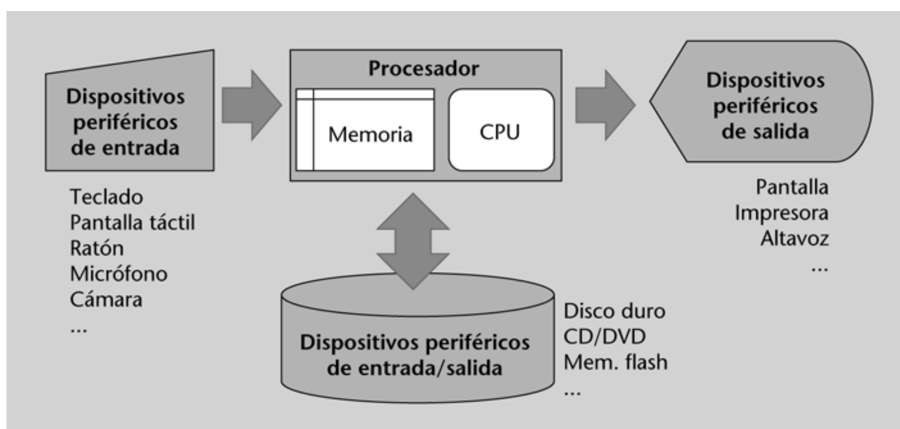
Un computador digital es un computador que trabaja con datos numéricos, cuya interpretación depende del formato con que se esté trabajando, codificados todos ellos en un sistema de numeración en base 2, es decir, basado en señales binarias, señales sobre las que podemos identificar sólo dos valores distintos. ❗

El concepto de computador es, en principio, independiente de la tecnología utilizada para construirlo. Es cierto, sin embargo, que en la actualidad los computadores digitales se basan en la electrónica digital y que, por lo tanto, un computador digital es un sistema digital complejo.

La complejidad que rodea un computador digital hace inviable su concepción sin una estructura y organización en módulos diferenciados con tareas y funcionalidades bien definidas. La estructura general de un computador digital es la que se representa de forma esquemática en la figura 1, donde el sentido de las flechas indica el flujo de información. Podríamos definir la ecuación de funcionamiento de la forma siguiente:

Datos de entrada + procesamiento = resultado (datos de salida)

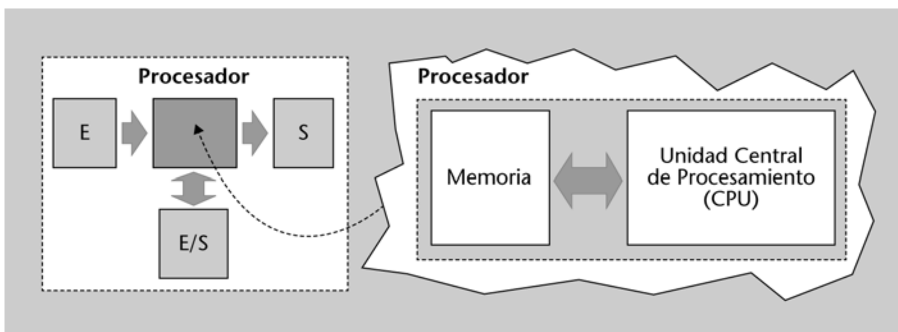
Figura 1. Estructura general de un computador



Los **dispositivos de entrada y los de salida** claramente constituyen elementos de conversión de la información entre el mundo analógico que nos rodea y el mundo digital en el que trabaja el procesador. Los dispositivos de entrada/salida, mayoritariamente, están constituidos por dispositivos para almacenar información digital, en uno u otro formato, pero información digital que el procesador puede recuperar.

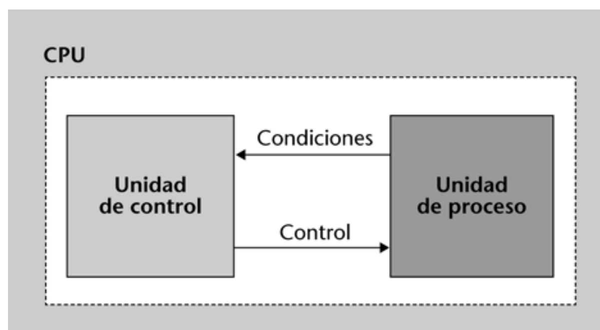
El **procesador** está constituido por una **unidad central de proceso** (CPU, *Central Process Unit*) y una **memoria** íntimamente relacionada con él (figura 2).

Figura 2. Arquitectura general de un procesador




La unidad central de proceso es realmente la encargada de procesar los datos de acuerdo con el programa establecido, y se organiza en dos grandes bloques, como se muestra en la figura 3, la **unidad de control** y la **unidad de proceso** o **camino de datos**.

Figura 3. Estructura de una CPU




La unidad de proceso reúne los recursos de cálculo, y la unidad de control es la encargada de dar las órdenes en la secuencia correcta a la unidad de proceso para realizar las operaciones que establece el programa en ejecución.

### 3.1. Arquitectura de Von Neuman

Se conoce por este nombre la arquitectura que implementan los computadores actuales y que se describe por primera vez en un documento escrito por John Von Neumann (1903-1957) como colaborador en el proyecto EDVAC, de donde toma el nombre.

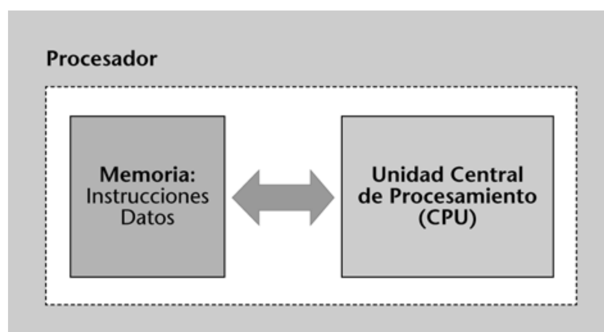
La característica distintiva es que se trata de una arquitectura en la que tanto los datos como el programa se almacenan en la memoria principal, que está ligada directamente a la CPU. El concepto de programa almacenado difiere radicalmente del tipo de programación que se practicaba en los computadores precedentes, y que se basaba en la modificación de los circuitos electrónicos.

Esta arquitectura es la base de los computadores modernos, en los que podemos identificar estas dos características:

**1) Programa almacenado.** Tanto los datos como las instrucciones del programa a ejecutar se encuentran en la memoria principal del computador. De este hecho se derivan dos consecuencias. Por una parte, esta característica dota al computador de una amplia generalidad. De la otra, la comunicación entre la memoria y la CPU se convierte en crítica y constituye un verdadero cuello de botella en el rendimiento de la máquina.

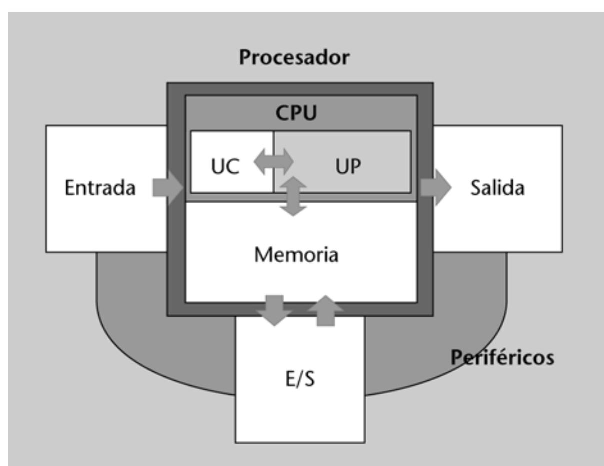
**2) Unidad de control (relativamente) simple**. En esta arquitectura, la unidad de control no se tiene que ocupar de ejecutar todo el programa, sino que hace de manera iterativa una única tarea: el ciclo de ejecución de instrucciones.

Figura 4. Arquitectura de Von Neumann




Con un procesador de este tipo, la estructura básica de un computador digital es la que aparece en la figura 5.

Figura 5. Arquitectura de un computador tipo Von Neumann



### 3.2. La arquitectura de Harvard

La arquitectura de Von Neumann tiene en sí misma dos grandes limitaciones. Por una parte, el acceso a memoria es un punto crítico y limita el rendimiento de los sistemas basados en este tipo de arquitectura. Por otra parte, lleva implícita la idea de la ejecución secuencial, es decir, de la ejecución de una única instrucción al mismo tiempo, lo cual limita las posibilidades de ejecución en paralelo.

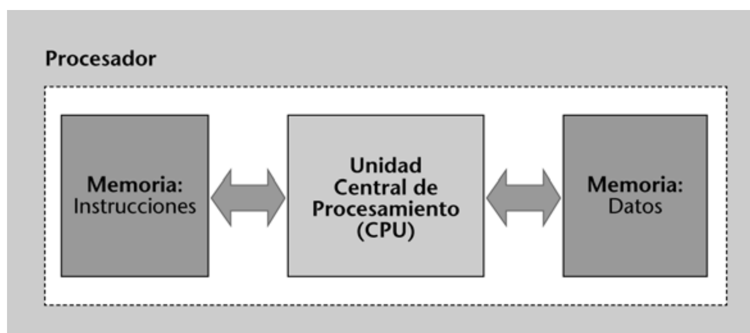
De las llamadas arquitecturas no Von Neumann podemos destacar la arquitectura de Harvard. La característica principal de esta arquitectura es que dispone de una memoria dedicada al programa y una segunda memoria para los datos. Esta diferencia ayuda a corregir la limitación que supone el acceso a memoria, ya que permite hacer operaciones con la memoria de datos mientras se accede a la memoria de programa. Por otra parte, también limita la posibilidad de la automodificación de los programas, que si bien desde el primer momento fue un aliciente en la arquitectura de Von Neumann, ha llegado a convertirse en un problema.

**Automodificación del código**

La automodificación del código (la capacidad de un programa para cambiarse a sí mismo) ha sido uno de los recursos que se ha aprovechado para elaborar código malintencionado como los virus.

El uso de la arquitectura de Harvard se ha extendido en el campo de los microcontroladores y de la electrónica distribuida. Su estructura general la podemos ver reflejada en la figura 6.

Figura 6. Arquitectura de Harvard



# Resumen

En este módulo se hace una introducción a los conceptos que se trabajan a lo largo de los módulos siguientes: la codificación adecuada de la información para interpretarla y tratarla dentro de los computadores, la tecnología con la que se construyen los sistemas digitales en general y la arquitectura básica de los computadores.

Los computadores actuales se presentan como el resultado de una evolución que se ha llevado a cabo a lo largo de los años, partiendo de la arquitectura básica fijada en la primera generación de computadores, con la incorporación del concepto de programa almacenado, y se enumeran los principales cambios tecnológicos que han permitido mejorar el rendimiento de las máquinas hasta el momento actual.

Se dedica un apartado a describir con más detalle la arquitectura básica de los computadores digitales, indicando la relación entre los dispositivos de entrada, el procesador y los dispositivos de salida. Al mismo tiempo, se muestran los elementos constitutivos del procesador: la CPU y la memoria.

# Bibliografía

**Augarten, S.** (1984). *Bit by Bit. An Illustrated History of Computers*. Nueva York: Ticknor & Fields

**Ceruzi, P. E.** (1998). *A History of Modern Computing*. Massachussets: The MIT Press.

**Williams, M. R.** (1997). *History of Computing Technology*. Los Alamitos, CA: IEEE Computer Society Press.

# Representación de la información

A. Josep Velasco González

Con la colaboración de:
Ramon Costa Castelló
Montse Peiron Guàrdia

PID_00163598

Universitat Oberta
de Catalunya

www.uoc.edu

# Índice

**Introducción** .......... 5

**Objetivos** .......... 7

**1. Los números y los sistemas de representación** .......... 9
1.1. Sistemas de representación .......... 9
1.2. Sistemas de numeración posicionales .......... 10
1.3. Cambios de base .......... 13
1.3.1. Método basado en el TFN .......... 13
1.3.2. Método basado en el teorema de la división entera .......... 14
1.3.3. Cambio de base entre $b$ y $b^n$ .......... 17
1.4. Empaquetamiento de la información .......... 18
1.5. Números con signo .......... 21
1.6. Suma en los sistemas posicionales .......... 22
1.7. Resta en los sistemas posicionales .......... 22
1.8. Multiplicación y división por potencias de la base de numeración .......... 23

**2. Representación de los números en un computador** .......... 26
2.1. Condicionantes físicos .......... 26
2.1.1. Rango de representación .......... 27
2.1.2. Precisión .......... 28
2.1.3. Error de representación .......... 28
2.1.4. Aproximaciones: truncamiento y redondeo .......... 28
2.1.5. Desbordamiento .......... 30
2.2. Números naturales .......... 31
2.3. Números enteros .......... 33
2.3.1. Representación de enteros en signo y magnitud en base 2 .......... 33
2.3.2. Suma y resta en signo y magnitud .......... 35
2.3.3. Representación en complemento a 2 .......... 36
2.3.4. Cambio de signo en complemento 2 .......... 38
2.3.5. Magnitud de los números en complemento a 2 .......... 40
2.3.6. Suma en complemento a 2 .......... 40
2.3.7. Resta en complemento a 2 .......... 42
2.3.8. Multiplicación por $2^k$ de números en complemento a 2 .......... 43
2.4. Números fraccionarios .......... 44

**3. Otros tipos de representaciones** .......... 53
3.1. Representación de información alfanumérica .......... 53
3.2. Codificación de señales analógicas .......... 55
3.3. Otras representaciones numéricas .......... 58

3.3.1. Representación en exceso a *M* .......... 58
3.3.2. Representación en coma flotante .......... 60
3.3.3. Representación BCD .......... 64

**Resumen** .......... 66

**Ejercicios de autoevaluación** .......... 67

**Solucionario** .......... 68

**Glosario** .......... 94

**Bibliografía** .......... 95

# Introducción

Inicialmente, los computadores fueron desarrollados como una herramienta para agilizar la realización repetitiva de operaciones aritméticas y lógicas básicas, que con el tiempo fueron ganando complejidad, tanto por el número de operaciones como por la complejidad propia de los cálculos. Hoy en día, sin haber perdido la utilidad original, los computadores se han ido diversificando, adaptándose a múltiples aplicaciones hasta convertirse en un elemento imprescindible en todos los campos de la ciencia, de la comunicación y del ocio.

A pesar de los grandes cambios que han ido sufriendo las máquinas, el procesamiento de los datos dentro de un computador continúa basado en la realización de operaciones aritméticas y lógicas sencillas sobre datos que se encuentran en la memoria principal. Allí pueden haber llegado de procedencias diversas, pero en todos los casos, la información ha sufrido una transformación: se ha codificado de manera adecuada para poder ser tratada por un procesador digital.

Las características de la tecnología con la que se construyen los computadores obligan a trabajar con sólo dos símbolos diferentes: el 0 y el 1. Toda la información que tenga que procesar un computador se tendrá que codificar usando únicamente estos dos símbolos.

Dentro de un computador, cualquier información (valor numérico, texto, audio, vídeo) está representada como una cadena de 0's y 1's. Ahora bien, una cadena de ceros y unos sólo tiene sentido si conocemos el formato de representación, es decir, la manera como está codificada la información, lo cual incluye saber: el tipo de dato (es un número, un texto, una señal de audio digitalizada, etc.) y el sistema utilizado para representar este tipo de datos (es decir, el sistema de numeración, si es un número; la tabla de codificación de los caracteres, si se trata de un texto; el algoritmo de codificación y/o compresión por información multimedia; etc.)

**Nota**

Se usan los símbolos 0 y 1, porque son los dígitos binarios, el sistema que emplean los computadores. Además, también se usan para designar los términos *verdad* y *falso* en las operaciones lógicas.

¿Qué codifica la cadena 10100100? Pues depende. ¿De qué tipo de dato se trata? Si es un texto, y se ha usado el código ASCII ISO-8859-15 se trata del carácter "**€**"; si es un número natural, se trata del valor decimal 164; si es un entero codificado en signo y magnitud, es el valor decimal –36; si es un entero codificado en el sistema de complemento a 2, es el valor decimal –92; etc. En todos los casos se trata de la misma cadena, pero en cada caso se está considerando que esta cadena es el resultado de codificar la información de una manera diferente.

**Nota**

Signo y magnitud y complemento a 2 son sistemas de representación de números con signo que se describen en la segunda sección de este módulo.

La información que procesa un computador digital está codificada en cadenas de ceros y unos, y esto quiere decir que las operaciones que tienen lugar en el

computador son operaciones sobre cadenas de ceros y unos. De hecho, todo el procesamiento que se hace en los computadores se reduce a operaciones aritméticas y lógicas sencillas sobre las cadenas que codifican la información.

Estos son, pues, los puntos de partida:

- Dentro de un computador toda la información se codifica como cadenas de ceros y unos.

- Una cadena de ceros y unos no tiene sentido por ella misma. Hay que conocer la manera como se codifica la información, esto es el formato en que están codificados los datos.

- El procesamiento que lleva a cabo un computador sobre las cadenas de ceros y unos consiste en operaciones aritméticas y lógicas sencillas.

Mayoritariamente, la información dentro de los computadores es tratada como números y operada como tal, por lo tanto, conocer la manera en que se codifican los números es básico para entender el funcionamiento de los computadores.

En este módulo se explican los sistemas básicos de codificación de la información, prestando especial atención a la representación de la información numérica, a la que se dedica la mayor parte del módulo. El módulo se estructura de la forma siguiente. En primer lugar, se hace un análisis del sistema de numeración con el que estamos habituados a trabajar. A continuación, se explican los sistemas de codificación de números más usuales en los computadores, y finalmente, se dan las pautas para la codificación de datos no numéricos.

# Objetivos

Se enumeran a continuación los principales objetivos que hay que lograr con el estudio de este módulo:

**1.** Comprender cómo se puede representar cualquier tipo de información dentro de los computadores y conocer los principios básicos de la codificación.

**2.** Conocer en profundidad los sistemas ponderados no redundantes de base fija 2, 10 y 16, además de saber representar un mismo valor numérico en bases diferentes.

**3.** Comprender y saber utilizar los formatos con que se codifica la información numérica en un computador: el sistema ponderado en binario para los números naturales; signo y magnitud y complemento a 2 para los números enteros, y la representación de números fraccionarios en coma fija.

**4.** Conocer las operaciones aritméticas básicas que lleva a cabo un computador y saber efectuarlas a mano. Estas operaciones son la suma, la resta y la multiplicación y división por potencias de la base de números naturales, enteros y fraccionarios.

**5.** Comprender los conceptos de rango y precisión de un formato de codificación de la información numérica en un computador, así como los conceptos de desbordamiento y de error de representación.

**6.** Entender la manera de empaquetar cadenas de unos y ceros a partir de la base 16.

**7.** Conocer la forma de representar caracteres en formato ASCII.

# 1. Los números y los sistemas de representación

El objetivo de esta sección es analizar el sistema de numeración que utilizamos, identificando los parámetros que lo definen. Para hacerlo, se introducen los conceptos de raíz o base y de peso asociado a la posición de un dígito. Seguidamente, se explican las técnicas para encontrar la representación de un número en un sistema posicional de raíz fija cuando se cambia la raíz. Finalmente, se hace un análisis de la operación más común, la suma, y de su homóloga, la resta, así como de la multiplicación y de la división de números por potencias de la base de numeración.

**Terminología**

A lo largo del texto utilizaremos indistintamente los términos *representar* y *codificar* para referirnos a la manera como se escribe un dato según una sintaxis y un conjunto de símbolos determinado.

## 1.1. Sistemas de representación

La idea de valor numérico es un concepto abstracto que determina una cantidad. Los valores numéricos están sujetos a un orden de precedencia que se utiliza para relacionarlos y llegar a conclusiones. Para trabajar de manera ágil con este tipo de información, tenemos que poder representar los valores numéricos de manera eficiente, por lo cual se han desarrollado los llamados **sistemas de numeración**.

> Un **sistema de numeración** es una metodología que permite representar un conjunto de valores numéricos.

El abanico de sistemas de numeración es bastante amplio. Entre otros, podemos encontrar los sistemas de raíz o base, los sistemas de dígitos firmados, los sistemas de residuos y los sistemas racionales. De estos, los sistemas basados en raíz (o base) son los que más se utilizan por las ventajas que aportan en la manipulación aritmética de los valores numéricos, y en ellos centraremos nuestra atención. !

**Terminología**

Se puede utilizar la designación de base o raíz de forma indistinta, a pesar de que es más común el uso de la palabra *base*: hablamos de sistemas de numeración en *base n*.

> Un **sistema de numeración basado en raíz** describe los valores numéricos en función de una o varias raíces. La raíz o base del sistema de numeración indica el número de dígitos diferentes de que se dispone.

Cuando trabajamos con base 10, disponemos de diez símbolos diferentes, que denominamos **dígitos**, para la representación: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Si la base del sistema de numeración es 2, se dispone de dos dígitos, el 0 y el 1. En base 16, hay dieciséis dígitos diferentes, que se representan por: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

**Terminología**

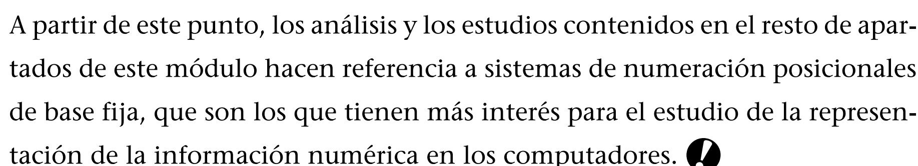
**Dígito**: cada uno de los signos gráficos empleados para representar los números en un sistema de numeración.

Los sistemas de numeración que usan sólo una base reciben el nombre de **sistemas de numeración de base fija**. El sistema de numeración que usamos en nuestra aritmética cotidiana es un sistema de numeración de base fija en que la base de numeración es 10.

**Los sistemas de numeración de base mixta**

Son los que usan más de una base de numeración. Un ejemplo de este tipo de sistema es el sistema horario, donde los valores vienen dados en función de las bases 24, 60 y 60 (horas, minutos y segundos).

Consideremos el número 321 en nuestro sistema de numeración en base 10. Hemos usado el dígito 3, el dígito 2 y el dígito 1, ordenados de una manera determinada. Estos mismos dígitos ordenados de otro modo (por ejemplo, 213) representan un número diferente, pese a estar constituido por los mismos dígitos. Los sistemas de numeración en los cuales el orden de los dígitos es determinante en la representación numérica se denominan **sistemas posicionales**.

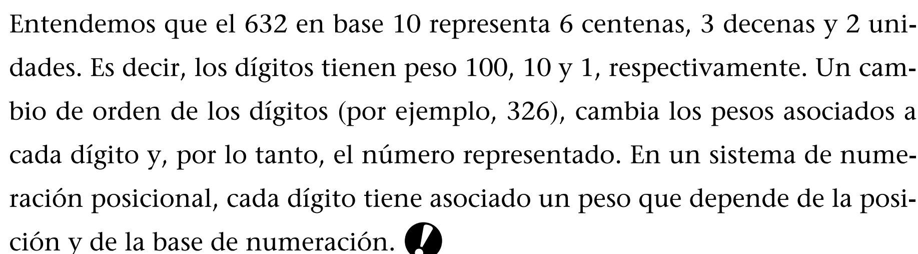


Un **sistema de numeración posicional** es aquél en que la representación de un valor numérico está determinada por una secuencia **ordenada** de dígitos.

A partir de este punto, los análisis y los estudios contenidos en el resto de apartados de este módulo hacen referencia a sistemas de numeración posicionales de base fija, que son los que tienen más interés para el estudio de la representación de la información numérica en los computadores.

## 1.2. Sistemas de numeración posicionales

Entendemos que el 632 en base 10 representa 6 centenas, 3 decenas y 2 unidades. Es decir, los dígitos tienen peso 100, 10 y 1, respectivamente. Un cambio de orden de los dígitos (por ejemplo, 326), cambia los pesos asociados a cada dígito y, por lo tanto, el número representado. En un sistema de numeración posicional, cada dígito tiene asociado un peso que depende de la posición y de la base de numeración.



Un **sistema de numeración posicional de base fija** es aquél en que un valor numérico $X$ se representa como una secuencia **ordenada** de dígitos, de la manera siguiente:

$$x_{n-1}x_{n-2}\cdots x_1x_0,x_{-1}\cdots x_{-m}$$

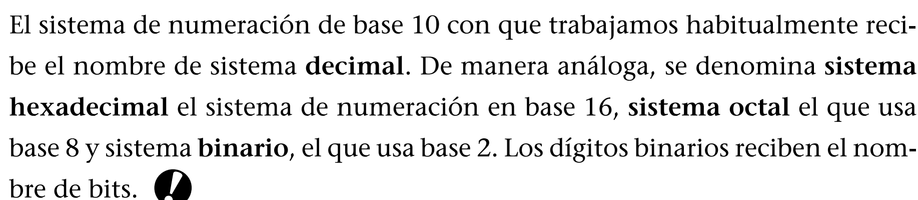
donde cada $x_i$ es un dígito tal que $0 \le x_i \le b-1$, donde $b$ es la base del sistema de numeración y $x_i$ es el dígito de la posición $i$-ésima de la secuencia.

**Terminología**

Utilizaremos $X$ para referirnos al concepto abstracto de valor numérico. La representación del valor numérico $X$ en base $b$ lo escribiremos de la forma $X_{(b}$ donde $b$ es la base en decimal.

Las posiciones con subíndice negativo corresponden a la **parte fraccionaria** del número, mientras que las posiciones con subíndice positivo corresponden a la **parte entera**. La frontera entre la parte entera y la parte fraccionaria se indica con una **coma**. Los dígitos de la parte entera se consignan a la izquierda de la coma y los de la parte fraccionaria a la derecha de la coma.

El sistema de numeración de base 10 con que trabajamos habitualmente recibe el nombre de sistema **decimal**. De manera análoga, se denomina **sistema hexadecimal** el sistema de numeración en base 16, **sistema octal** el que usa base 8 y sistema **binario**, el que usa base 2. Los dígitos binarios reciben el nombre de bits.



**Terminología**

Evitaremos utilizar la expresión *parte decimal*, para designar la parte fraccionaria de un número y eludiremos, así, la ambigüedad del término *número decimal*. Un **número decimal** es un número en base 10, no un número con parte fraccionaria.

Consideremos, de nuevo, el número $632_{(10}$. Lo podemos escribir en función de los pesos asociados a cada posición:

$$632_{(10} = 6 \cdot 100 + 3 \cdot 10 + 2 \cdot 1$$

**Terminología**

Un dígito binario recibe el nombre de *bit*, que es un acrónimo de la expresión inglesa *binary digit*.

Según la definición, el 2 ocupa la posición 0, el 3 la posición 1 y el 6 la posición 2. Podemos reescribir la expresión anterior relacionando los pesos con la base de numeración y con la posición que ocupa cada dígito:

$$632_{(10} = 6 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0$$

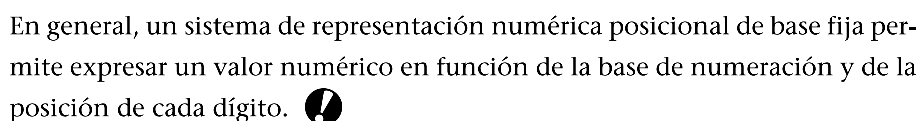
El $34{,}75_{(10}$ también se puede escribir en función de la base y de las posiciones:

$$34{,}75_{(10} = 3 \cdot 10^1 + 4 \cdot 10^0 + 7 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

**Nota**

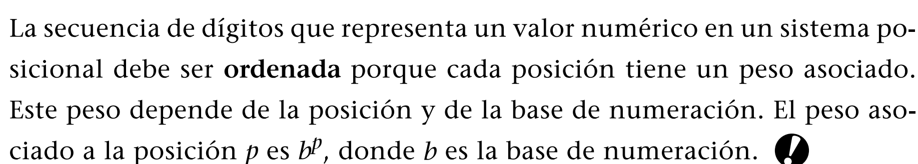
Según la numeración de posiciones definida, el 7 ocupa la posición –1 y el 5 la posición –2, mientras que el 3 y el 4 (dígitos de la parte entera) ocupan las posiciones 1 y 0, respectivamente.

En general, un sistema de representación numérica posicional de base fija permite expresar un valor numérico en función de la base de numeración y de la posición de cada dígito.



**Recordemos que** $x^{-k} = \frac{1}{x^k}$

La secuencia de dígitos que representa un valor numérico en un sistema posicional debe ser **ordenada** porque cada posición tiene un peso asociado. Este peso depende de la posición y de la base de numeración. El peso asociado a la posición $p$ es $b^p$, donde $b$ es la base de numeración.



El número $X$ representado por la secuencia de dígitos $x_{n-1}x_{n-2}\cdots x_1x_0,x_{-1}\cdots x_{-m}$ se puede expresar en función de la base de numeración de la forma:

$$X = \sum_{i=-m}^{n-1} x_i\, b^i = x_{n-1}\cdot b^{n-1} + x_{n-2}\cdot b^{n-2} + \cdots + x_{-m}\cdot b^{-m}$$

donde cada $x_i$ es un dígito tal que $0 \le x_i \le b-1$, donde $b$ es la base del sistema de numeración y $x_i$ el dígito de la posición $i$-ésima de la secuencia.

Esta expresión se conoce como el **teorema fundamental de la numeración (TFN).***

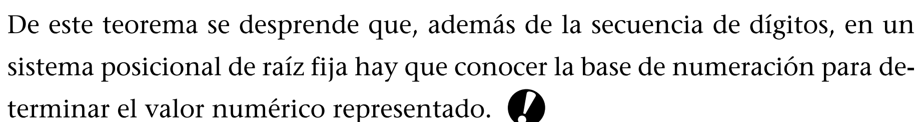
**Terminología**

Expresar un número en función de la base de numeración equivale a escribirlo de la forma:

$$x_{n-1}\cdot b^{n-1} + x_{n-2}\cdot b^{n-2} + \cdots$$

*** Abreviaremos *teorema fundamental de la numeración* con la sigla TFN.**

De este teorema se desprende que, además de la secuencia de dígitos, en un sistema posicional de raíz fija hay que conocer la base de numeración para determinar el valor numérico representado.



La secuencia de dígitos 235 es válida en todas las bases más grandes que 5 (porque el 5 no es un dígito válido en bases inferiores a 6). Ahora bien, en bases diferentes representa números diferentes. Por lo tanto, $235_{(6} \neq 235_{(10} \neq 235_{(16}$. La tabla siguiente muestra la correspondencia entre las representaciones de algunos valores:

| Base 2 | Base 4 | Base 8 | Base 10 | Base 16 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 10 | 2 | 2 | 2 | 2 |
| 11 | 3 | 3 | 3 | 3 |
| 100 | 10 | 4 | 4 | 4 |
| 101 | 11 | 5 | 5 | 5 |
| 110 | 12 | 6 | 6 | 6 |
| 111 | 13 | 7 | 7 | 7 |
| 1000 | 20 | 10 | 8 | 8 |
| 1001 | 21 | 11 | 9 | 9 |
| 1010 | 22 | 12 | 10 | A |
| 1011 | 23 | 13 | 11 | B |
| 1100 | 30 | 14 | 12 | C |
| 1101 | 31 | 15 | 13 | D |
| 1110 | 32 | 16 | 14 | E |
| 1111 | 33 | 17 | 15 | F |
| 10000 | 100 | 20 | 16 | 10 |
| 10001 | 101 | 21 | 17 | 11 |
| 10010 | 110 | 22 | 18 | 12 |

**Elementos de la tabla**

En cada columna se representan los valores numéricos desde el 0 hasta el $18_{(10}$ en la base indicada en la casilla superior de la columna. En cada fila disponemos de la representación del mismo valor numérico en diferentes bases.

### 1.3. Cambios de base

La secuencia ordenada de dígitos que representa un valor numérico cambia según la base del sistema de numeración, pero hay una relación entre las secuencias de dígitos.

Los **métodos de cambio de base** permiten encontrar la secuencia ordenada de dígitos que representa un valor numérico *X* en el sistema de numeración en base *b'*, a partir de la representación en el sistema de numeración en base *b*, es decir:

$$canvi_a_base_b'\left(X_{(b}\right) = X_{(b'}$$

**Uso de los cambios de base**

Utilizaremos los cambios de base para convertir la representación de un número entre las bases 2, 10 y 16.

En los apartados siguientes, se exponen dos técnicas de cambio de base.

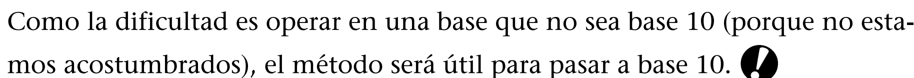
#### 1.3.1. Método basado en el TFN

Si aplicamos el TFN al $324_{(10}$ lo podemos escribir como:

$$324_{(10} = 3 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0$$

Haciendo las operaciones de la derecha en base 10, se obtiene la representación en base 10, que es la que tenemos a la izquierda de la igualdad. Ahora bien, si hacemos las operaciones en base 7, tendremos la representación en base 7. En general, si hacemos las operaciones en base *b* obtenemos la representación en base *b*.

Como la dificultad es operar en una base que no sea base 10 (porque no estamos acostumbrados), el método será útil para pasar a base 10.



| Valores decimales | Dígitos Hexadecimales |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | A |
| 11 | B |
| 12 | C |
| 13 | D |
| 14 | E |
| 15 | F |

**Cambio de base basado en el TFN**

Para cambiar a base 10 el $462_{(7}$:

**1)** Expresamos el número en función de la base (base 7) según el TFN:

$$462_{(7} = 4 \cdot 7^2 + 6 \cdot 7^1 + 2 \cdot 7^0$$

**2)** Hacemos las operaciones en la base de llegada (base 10):

$$4 \cdot 7^2 + 6 \cdot 7^1 + 2 \cdot 7^0 = 4 \cdot 49 + 6 \cdot 7 + 2 \cdot 1 = 240_{(10}$$

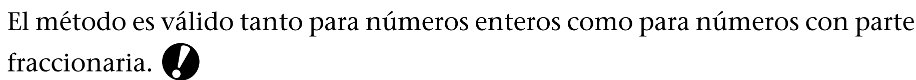
Las secuencias de dígitos $462_{(7}$ y $240_{(10}$ representan el mismo valor numérico, pero en bases diferentes: base 7 la primera y base 10 la segunda.

Para hallar la representación de $X_{(b}$ en base 10 tenemos que:

**1)** Expresar $X_{(b}$ en función de la base *b*, siguiendo el TFN;
**2)** Hacer las operaciones en base 10.

Cuando $b > 10$ los dígitos de la base *b* se tienen que cambiar a base 10 antes de hacer las operaciones.

El método es válido tanto para números enteros como para números con parte fraccionaria.



**Cambios de base basados en el TFN**

Para pasar a base 10 el número $101100{,}01_{(2}$:

**1)** Expresamos el número en función de la base (base 2):

$$101100{,}01_{(2} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

**2)** Hacemos las operaciones en base 10:

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} =$$
$$1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 + 0 \cdot 0{,}5 + 1 \cdot 0{,}25 = 44{,}25_{(10}$$

El $101100{,}01_{(2}$ en base 10 es el $44{,}25_{(10}$.

Para pasar a base 10 el número $AF2C{,}2_{(16}$:

**1)** Expresamos el número en función de la base (base 16):

$$AF2C{,}2_{(16} = A \cdot 16^3 + F \cdot 16^2 + 2 \cdot 16^1 + C \cdot 16^0 + 2 \cdot 16^{-1}$$

**2)** Hacemos las operaciones en base 10. En este caso, tenemos que cambiar a base 10 los dígitos hexadecimales antes de hacer las operaciones:

$$A \cdot 16^3 + F \cdot 16^2 + 2 \cdot 16^1 + C \cdot 16^0 + 2 \cdot 16^{-1} =$$
$$10 \cdot 16^3 + 15 \cdot 16^2 + 2 \cdot 16^1 + 12 \cdot 16^0 + 2 \cdot 16^{-1} = 44844{,}125_{(10}$$

El $AF2C{,}2_{(16}$ en base 10 es el $44844{,}125_{(10}$.

### 1.3.2. Método basado en el teorema de la división entera

Este método de cambio de base consiste en hacer divisiones enteras por la nueva base de numeración de manera iterativa. Los residuos de las divisiones enteras son los dígitos de la representación en la nueva base.

Para cambiar a base 7 el número $317_{(10}$, hacemos divisiones enteras por 7:

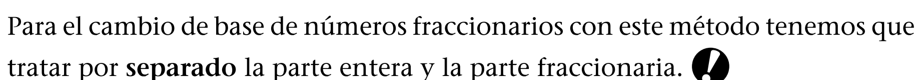
$$\begin{aligned} 317 &= 45 \cdot 7 + 2 \\ 45 &= 6 \cdot 7 + 3 \\ 6 &= 0 \cdot 7 + 6 \end{aligned} \quad \uparrow$$

$317_{(10} = 632_{(7}$

La secuencia de residuos en **orden inverso** nos da la representación en la nueva base. El número $317_{(10}$ en base 7 es el $632_{(7}$.

Como las operaciones se hacen en la base inicial, este método es especialmente útil para pasar de base 10 a otra.

Para el cambio de base de números fraccionarios con este método tenemos que tratar por **separado** la parte entera y la parte fraccionaria.



Para hallar la representación de $X_{(10}$ a base $b$:

**1) Parte entera:** sucesivamente, hacer en base 10 la división entera por la nueva base $b$. Paramos la sucesión de divisiones cuando obtenemos un cociente 0. La secuencia de residuos, tomados del último al primero, es la secuencia de dígitos de izquierda a derecha de la parte entera en la nueva base. Cuando $b > 10$, los residuos se tienen que pasar a dígitos de la nueva base.

**2) Parte fraccionaria:** sucesivamente, se separa la parte fraccionaria y se multiplica por la nueva base $b$. Las operaciones se hacen en base 10. Paramos la sucesión de multiplicaciones cuando encontramos un comportamiento periódico o cuando tenemos dígitos suficientes. La secuencia de valores enteros obtenidos al hacer las multiplicaciones tomados del primero al último es la secuencia de dígitos de izquierda a derecha en la nueva base de representación. Cuando $b > 10$ los enteros obtenidos se tienen que pasar a dígitos de la nueva base.

Finalmente, hay que unir la parte entera y la parte fraccionaria obtenidas.

**Ejemplo**

Un número con parte fraccionaria finita no periódica en una base puede tener una parte fraccionaria infinita periódica en otra base. Por ejemplo,

$$0,3_{(10} = 0,0100\widehat{11001}_{(2}.$$

**Cambios de base por el método de la división entera**

Cambiar a base 2 el $44,25_{(10}$:

**a) Parte entera:** sucesivamente, hacemos divisiones enteras por la nueva base (base 2) hasta obtener un cociente 0, y tomamos los residuos en orden inverso:

$$\begin{aligned} 44 &= 22 \cdot 2 + 0 \\ 22 &= 11 \cdot 2 + 0 \\ 11 &= 5 \cdot 2 + 1 \\ 5 &= 2 \cdot 2 + 1 \\ 2 &= 1 \cdot 2 + 0 \\ 1 &= 0 \cdot 2 + 1 \end{aligned} \uparrow$$

$44_{(10} = 101100_{(2}$

**b) Parte fraccionaria:** sucesivamente, multiplicamos por la nueva base (base 2):

$$\begin{aligned} 0,25 \cdot 2 &= 0,50 = 0,50 + 0 \\ 0,50 \cdot 2 &= 1,00 = 0,00 + 1 \end{aligned} \downarrow$$

$0,25_{(10} = 0,01_{(2}$

Para completar el cambio de base, unimos la parte entera y la parte fraccionaria que resultan:

$$44,25_{(10} = 101100_{(2} + 0,01_{(2} = 101100,01_{(2}$$

El $44,25_{(10}$ en base 2 es el $101100,01_{(2}$.

Cambiar a base 16 el $44844,12_{(10}$:

**a) Parte entera:** sucesivamente, hacemos divisiones enteras por 16 hasta obtener un cociente 0:

$$\begin{aligned} 44844 &= 2802 \cdot 16 + 12 \\ 2802 &= 175 \cdot 16 + 2 \\ 175 &= 10 \cdot 16 + 15 \\ 10 &= 0 \cdot 16 + 10 \end{aligned} \uparrow$$

Como la nueva base es mayor que 10, tenemos que convertir los residuos a la nueva base (base 16):

$$\begin{array}{r} 12_{(10} = C_{(16} \\ 2_{(10} = 2_{(16} \\ 15_{(10} = F_{(16} \\ 10_{(10} = A_{(16} \end{array} \qquad 44844_{(10} = AF2C_{(16}$$

**b) Parte fraccionaria:** sucesivamente, multiplicamos por 16:

$$\begin{array}{l} 0{,}12 \cdot 16 = \phantom{1}1{,}92 = 0{,}92 + \phantom{1}1 \\ 0{,}92 \cdot 16 = 14{,}72 = 0{,}72 + 14 \\ 0{,}72 \cdot 16 = 11{,}52 = 0{,}52 + 11 \\ 0{,}52 \cdot 16 = \phantom{1}8{,}32 = 0{,}32 + \phantom{1}8 \\ 0{,}32 \cdot 16 = \phantom{1}5{,}12 = 0{,}12 + \phantom{1}5 \\ 0{,}12 \cdot 16 = \phantom{1}1{,}92 = 0{,}92 + \phantom{1}1 \\ 0{,}92 \cdot 16 = 14{,}72 = 0{,}72 + 14 \\ \ldots \end{array} \downarrow$$

La secuencia de enteros que obtenemos se repite (1,14,11,8,5,1,14,...). Por lo tanto, es un número periódico. Además, los enteros se tienen que convertir a dígitos de base 16:

$$\begin{array}{r} 1_{(10} = 1_{(16} \\ 14_{(10} = E_{(16} \\ 11_{(10} = B_{(16} \\ 8_{(10} = 8_{(16} \\ 5_{(10} = 5_{(16} \\ 1_{(10} = 1_{(16} \end{array}$$

$$0{,}12_{(10} = 0{,}1EB851EB851EB\ldots_{(16} = 0{,}\widehat{1EB85}_{(16}$$

Finalmente, uniremos la parte entera y la parte fraccionaria:

$$44844{,}12_{(10} = AF2C{,}\widehat{1EB85}_{(16}$$

El $44844{,}12_{(10}$ en base 16 es el $AF2C{,}\widehat{1EB85}_{(16}$ .

Para cambiar de base *b* a base *b'*, donde ni *b* ni *b'* son la base 10, utilizamos la base 10 como base intermedia. Así, usamos el primer método (basado en el TFN) para pasar de base *b* a base 10 y, posteriormente, el segundo método (basado en el teorema de la división entera) para pasar de base 10 a base *b'*.

**Cambio entre bases diferentes de la base 10**

El cambio a base 6 del $232{,}1_{(4}$ lo tenemos que hacer en dos pasos:

**1)** Hacemos el cambio a base 10 del $232{,}1_{(4}$ aplicando el método del TFN:

$$\begin{array}{l} 232{,}1_{(2} = 2 \cdot 4^2 + 3 \cdot 4^1 + 2 \cdot 4^0 + 1 \cdot 4^{-1} = \\ \quad = 32 + 12 + 2 + 0{,}25 = 46{,}25_{(10} \end{array}$$

**2)** Hacemos el cambio a base 6 del $46{,}25_{(10}$:

$$\begin{array}{r} 46 = 7 \cdot 6 + 4 \\ 7 = 1 \cdot 6 + 1 \\ 1 = 0 \cdot 6 + 1 \end{array} \uparrow \qquad \begin{array}{l} 0{,}25 \cdot 6 = 1{,}50 = 0{,}50 + 1 \\ 0{,}50 \cdot 6 = 3{,}00 = 0{,}00 + 3 \end{array} \downarrow$$

$$46_{(10} = 114_{(6} \qquad 0{,}25_{(10} = 0{,}13_{(6}$$

El $46{,}25_{(10}$ es equivalente al $114{,}13_{(6}$

Por lo tanto, el $232{,}1_{(4}$ es el $114{,}13_{(6}$ en base 6.

| Potencias de 2 | |
|---|---|
| $2^{16}$ | 65536 |
| $2^{15}$ | 32768 |
| $2^{14}$ | 16384 |
| $2^{13}$ | 8192 |
| $2^{12}$ | 4096 |
| $2^{11}$ | 2048 |
| $2^{10}$ | 1024 |
| $2^{9}$ | 512 |
| $2^{8}$ | 256 |
| $2^{7}$ | 128 |
| $2^{6}$ | 64 |
| $2^{5}$ | 32 |
| $2^{4}$ | 16 |
| $2^{3}$ | 8 |
| $2^{2}$ | 4 |
| $2^{1}$ | 2 |
| $2^{0}$ | 1 |
| $2^{-1}$ | 0,5 |
| $2^{-2}$ | 0,25 |
| $2^{-3}$ | 0,125 |
| $2^{-4}$ | 0,0625 |
| $2^{-5}$ | 0,03125 |
| $2^{-6}$ | 0,015625 |
| $2^{-7}$ | 0,0078125 |
| $2^{-8}$ | 0,00390625 |

### 1.3.3. Cambio de base entre *b* y $b^n$

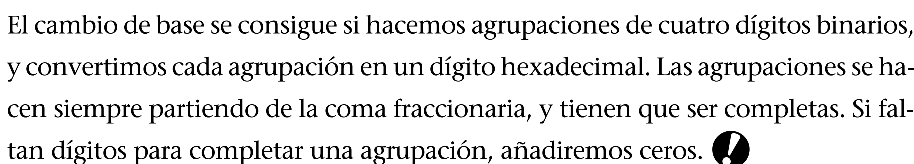
El cambio de base *b* a base $b^n$ es directo, porque...

> ...un dígito en base $b^n$ se corresponde con *n* dígitos en base *b*.

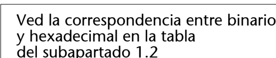
Esta circunstancia se da entre base 2 y base 16 (porque $16 = 2^4$) o entre base 16 y base 4 (porque $16 = 4^2$), pero no entre base 8 y base 16, porque 16 no es potencia de 8.

**Cambio de base *b* a base $b^n$**

En el cambio a base 16 del $10010110{,}01101101_{(2}$, tendremos en cuenta que 16 es potencia de 2: $16 = 2^4$. Esta relación indica que cada dígito de base 16 se corresponde con cuatro dígitos de base 2.



El cambio de base se consigue si hacemos agrupaciones de cuatro dígitos binarios, y convertimos cada agrupación en un dígito hexadecimal. Las agrupaciones se hacen siempre partiendo de la coma fraccionaria, y tienen que ser completas. Si faltan dígitos para completar una agrupación, añadiremos ceros.



Ved la correspondencia entre binario y hexadecimal en la tabla del subapartado 1.2

| | | | | | |
|---|---|---|---|---|---|
| 1001 | 0110 | , | 0110 | 1101 | (2 |
| 9 | 6 | , | 6 | D | (16 |

El $10010110{,}01101101_{(2}$ es en base 16 el $96{,}6D_{(16}$.

En el cambio a base 16 del $101110{,}101101_{(2}$ tenemos que completar las agrupaciones añadiendo ceros (en este caso, tanto en la parte entera como la fraccionaria):

| | | | | | |
|---|---|---|---|---|---|
| **00**10 | 1110 | , | 1011 | 01**00** | (2 |
| 2 | E | , | B | 4 | (16 |

$$101110{,}101101_{(2} = 2E{,}B4_{(16}$$

**Cambio de base $b^n$ a base *b***

Cuando el cambio es de base $b^n$ a *b*, el procedimiento es análogo pero en sentido inverso: cada dígito en base $b^n$ se transforma en *n* dígitos en base *b*.

Para cambiar a base 2 el $7632{,}13_{(8}$, tendremos en cuenta que $8 = 2^3$. Por consiguiente, cada dígito en base 8 dará lugar a tres dígitos binarios:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 3 | 2 | , | 1 | 3 | (8 |
| 111 | 110 | 011 | 010 | , | 001 | 011 | (2 |

$$7632{,}13_{(8} = 111110011010{,}001011_{(2}$$

Debemos prestar atención al hecho de que hay que obtener exactamente *n* dígitos en base *b* por cada dígito en base $b^n$ (en este caso, tres dígitos binarios por cada dígito octal), añadiendo para cada dígito los ceros necesarios. Veamos cómo en el cambio a base 2 del $E1B2,4F_{(16}$ cada dígito hexadecimal da lugar a cuatro dígitos binarios.

| E | 1 | B | 2 | , | 4 | F | (16 |
|---|---|---|---|---|---|---|---|
| 1110 | 0001 | 1011 | 0010 | , | 0100 | 1111 | (2 |

$$E1B2,4F_{(16} = 1110000110110010,01001111_{(2}$$

**Errores frecuentes**

A menudo se cometen dos errores en estos tipos de cambio de base:

**1)** Cuando hacemos un cambio de base $b^n$ a base *b*, cada dígito de base $b^n$ tiene que dar lugar, exactamente, a *n* dígitos en base *b*. Hay que evitar el error siguiente:

$$A3_{(16} = 101011_{(2}$$

donde el dígito A ha dado lugar a los bits 1010 y el dígito 3 a los bits 11. En realidad, ha de ser:

$$A3_{(16} = 1010\mathbf{00}11_{(2}$$

donde se han añadido dos ceros para completar el conjunto de cuatro dígitos que debe generar el dígito hexadecimal 3.

**2)** Cuando hacemos un cambio de base *b* a base $b^n$, son necesarios *n* dígitos de base *b* para obtener un dígito en base $b^n$. Hay que evitar el error siguiente:

$$1100,11_{(2} = C,3_{(16}$$

donde los bits 1100 dan lugar al dígito hexadecimal C y los bits 11 al 3. En realidad, ha de ser:

$$1100,11\mathbf{00}_{(2} = C,C_{(16}$$

donde se han añadido 2 ceros a la derecha con objeto de constituir un grupo de cuatro dígitos binarios que dan lugar al dígito hexadecimal C.

### 1.4. Empaquetamiento de la información

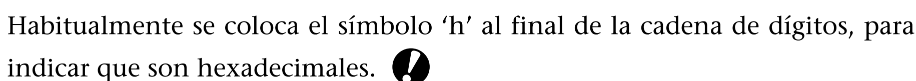
Con los cambios a base 2 tenemos un camino abierto para procesar los números dentro de los computadores. De hecho, dentro de los computadores, toda la in-

formación (no sólo la numérica) se codifica utilizando únicamente el símbolo 1 y el símbolo 0. Por lo tanto, toda la información que procesa un computador está codificada en cadenas de unos y de ceros, es decir, en cadenas de bits.

Ahora bien, disponer sólo de dos símbolos nos lleva a representaciones con un gran número de dígitos, a cadenas de bits largas, que para nosotros (no para los computadores) son difíciles de recordar y de manipular.

Pues bien, podemos aprovechar la técnica de hacer agrupaciones de cuatro bits (en vista de los cambios de base 2 a base 16) para convertir las cadenas de bits en dígitos hexadecimales y compactarlas, así, en cadenas mucho más cortas y manejables. Este proceso recibe el nombre de **empaquetamiento hexadecimal**. El proceso inverso se denomina **desempaquetamiento**.

> El **empaquetamiento hexadecimal** consiste en compactar información binaria en cadenas de dígitos hexadecimales.



Habitualmente se coloca el símbolo 'h' al final de la cadena de dígitos, para indicar que son hexadecimales.

**Empaquetamiento de una cadena de bits**

Para empaquetar la cadena de bits 110100100011, procedemos de la forma siguiente:

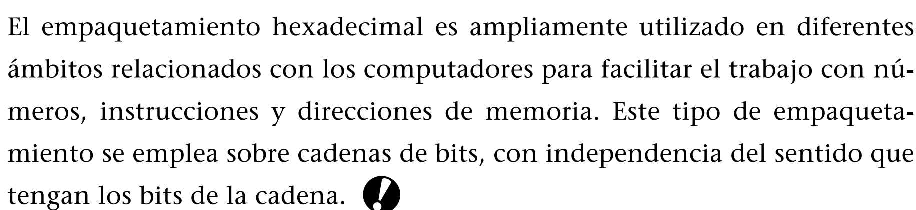
**1)** Dividimos la cadena 110100100011 de derecha a izquierda en grupos de 4 bits:

1101 – 0010 – 0011

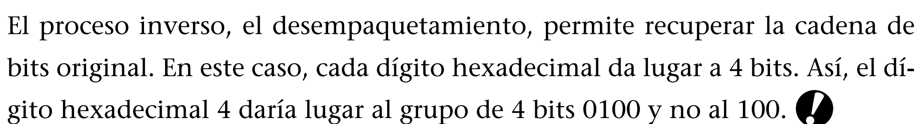
**2)** Codificamos cada grupo de 4 bits como un dígito hexadecimal:

1101– 0010 – 0011
D – 2 – 3

Por lo tanto, si hacemos el empaquetamiento hexadecimal de la cadena de bits 110100100011, se obtiene D23h.



El empaquetamiento hexadecimal es ampliamente utilizado en diferentes ámbitos relacionados con los computadores para facilitar el trabajo con números, instrucciones y direcciones de memoria. Este tipo de empaquetamiento se emplea sobre cadenas de bits, con independencia del sentido que tengan los bits de la cadena.



El proceso inverso, el desempaquetamiento, permite recuperar la cadena de bits original. En este caso, cada dígito hexadecimal da lugar a 4 bits. Así, el dígito hexadecimal 4 daría lugar al grupo de 4 bits 0100 y no al 100.

**Desempaquetamiento**

Para desempaquetar la cadena D23h, convertimos los dígitos hexadecimales a base 2 usando 4 bits para cada uno:

D – 2 – 3
1101– 0010 – 0011

Por lo tanto, si desempaquetamos la cadena de dígitos hexadecimales D23h, se obtiene la cadena de bits 110100100011.

Es importante diferenciar el concepto de empaquetamiento hexadecimal de cadenas de bits del concepto del cambio de base 2 a base 16. Cuando hagamos un cambio de base 2 a base 16 de un número, debemos tener presente la posición de la coma fraccionaria, porque buscamos la representación del mismo número pero en base 16. Por lo tanto, las agrupaciones de 4 bits se hacen a partir de la coma fraccionaria: hacia la izquierda de la coma, para obtener los dígitos hexadecimales enteros y hacia la derecha para conseguir los dígitos hexadecimales fraccionarios. En cambio, en el empaquetamiento hexadecimal no se tiene en cuenta el sentido de la información codificada y los bits se agrupan de 4 en 4 de derecha a izquierda independientemente de su sentido. En este caso, lo que obtenemos finalmente no es la representación del número en base 16, sino una cadena de dígitos hexadecimales que codifican una cadena de bits.

Veamos esta diferencia según hagamos el cambio a base 16 del número $111010{,}11_{(2}$ o el empaquetamiento hexadecimal. Si queremos hacer el cambio a base 16, tenemos que hacer agrupaciones a partir de la coma fraccionaria, añadiendo los ceros necesarios para completar las agrupaciones tanto por la derecha como por la izquierda:

$$\begin{array}{ccccc} \mathbf{00}11 & 1010 & , & 11\mathbf{00} & _{(2} \\ 3 & A & , & C & _{(16} \end{array}$$

En este caso, el resultado que se obtiene indica que el número $111010{,}11_{(2}$ en base 16 es el $3A{,}C_{(16}$.

En cambio, si queremos hacer un empaquetamiento hexadecimal, las agrupaciones se hacen de derecha a izquierda, sin tener en cuenta la posición de la coma. Se trata como una tira de unos y ceros. El resultado final no guarda información sobre la coma fraccionaria:

$$\begin{array}{cc} 1110 & 10{,}11_{(2} \\ E & B \end{array}$$

En este segundo caso, el resultado que se obtiene indica que el empaquetamiento hexadecimal de la cadena de bits 11101011 es EBh. Podemos comprobar que la secuencia de dígitos hexadecimales que se obtiene en uno y otro caso puede ser diferente.

### Actividades

**1.** Convertid a base 10 los valores siguientes:

**a)** $10011101_{(2}$
**b)** $3AD_{(16}$
**c)** $333_{(4}$

**d)** $333_{(8}$
**e)** $B2,3_{(16}$
**f)** $3245_{(8}$
**g)** $AC3C_{(16}$
**h)** $1010,11_{(8}$
**i)** $110011,11_{(4}$
**j)** $10011001,1101_{(2}$
**k)** $1110100,01101_{(2}$

**2.** Convertid a base 2 los valores siguientes:
**a)** $425_{(10}$
**b)** $344_{(10}$
**c)** $31,125_{(10}$
**d)** $4365,14_{(10}$

**3.** Convertid a hexadecimal los números siguientes:
**a)** $111010011,1110100111_{(2}$
**b)** $0,1101101_{(2}$
**d)** $45367_{(10}$
**c)** $111011,1010010101_{(2}$

**4.** Convertid los números hexadecimales siguientes a base 2, base 4 y base 8:
**a)** $ABCD_{(16}$
**b)** $45,45_{(16}$
**c)** $96FF,FF_{(16}$

**5.** Rellenad la tabla siguiente:

| Binario | Octal | Hexadecimal | Decimal |
|---|---|---|---|
| **1101100,110** | | | |
| | **362,23** | | |
| | | **A1,03** | |
| | | | **74,3** |

En cada fila veréis un valor numérico expresado en la base que indica la casilla superior de la columna donde se encuentra. Consignad en el resto de casillas la representación correspondiente según la base indicada en la parte superior.

**6.** Empaquetad en hexadecimal la cadena de bits 10110001.

**7.** Empaquetad en hexadecimal el número $0100000111,111010_{(2}$ que está en un formato de coma fija de 16 bits, de los cuales 6 son fraccionarios.

**8.** Desempaquetad la cadena de bits A83h y,
**a)** Encontrad el valor decimal si se trata de un número natural.
**b)** Encontrad el valor decimal si se trata de un número en coma fija sin signo de 12 bits, donde 4 son fraccionarios.

**9.** Consideremos el número $1010,101_{(2}$.
**a)** Haced el cambio a base 16.
**b)** Haced el empaquetamiento hexadecimal.

### 1.5. Números con signo

Cuando representamos magnitudes, a menudo les asignamos un signo (+/–) que precede a la magnitud y que indica si la magnitud es positiva o negativa. El símbolo – identifica las magnitudes negativas y el símbolo + las positivas:

$+23_{(10}$ $\quad$ $-456_{(8}$

$-34,5_{(7}$ $\quad$ $+AF,34_{(16}$

**Signo (+/–)**

A veces, cuando se trabaja con números con signo, el signo positivo (+) no se escribe y sólo aparece el signo cuando se trata de un número negativo.

Designaremos los números que llevan la información de signo como números con signo, en contraposición a los números sin signo, que sólo nos dan información sobre la magnitud del valor numérico.

## 1.6. Suma en los sistemas posicionales

El algoritmo de suma de dos números decimales que estamos habituados a utilizar progresa de derecha a izquierda, sumando en cada etapa los dígitos del mismo peso (los que ocupan la misma posición). Si la suma de estos dígitos llega al valor de la base (10 en este caso), genera un **acarreo** (lo que nos "llevamos") que se sumará con los dígitos de la etapa siguiente:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | | 1 | | | | ← dígito de acarreo* |
| | | 8 | 3 | 4 | 1 | (10 | |
| + | | 2 | 4 | 6 | 3 | (10 | |
| | 1 | 0 | 8 | 0 | 4 | (10 | ← resultado |

* El dígito de acarreo recibe en inglés el nombre de *carry*. Este término es de uso habitual en el entorno de los computadores.

En hexadecimal, se siguen las mismas pautas de suma, pero teniendo en cuenta que hay 16 dígitos diferentes:

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | | | | ← acarreo |
| | 3 | 5 | 8 | 2 | (16 | |
| + | A | F | 1 | 8 | (16 | |
| | E | 4 | 9 | A | (16 | ← resultado |

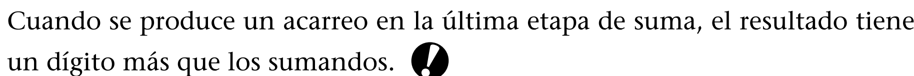
El proceso de suma en base 2 es análogo:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | 1 | | | | ← acarreo |
| | | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 (2 | |
| + | | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 (2 | |
| | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 (2 | ← resultado |

**Tabla de suma en base 2 acarreo / bit de suma**

| + | 0 | 1 |
|---|---|---|
| 0 | 0/0 | 0/1 |
| 1 | 0/1 | 1/0 |

Cuando se produce un acarreo en la última etapa de suma, el resultado tiene un dígito más que los sumandos.



## 1.7. Resta en los sistemas posicionales

La operación de resta también se lleva a cabo de derecha a izquierda, operando los dígitos de igual peso, y considerando el acarreo* de la etapa precedente. La

* En inglés el acarreo en el caso de la resta recibe el nombre de *borrow* .

particularidad en esta operación es que el número de menor magnitud (sustraendo) es el que hay que restar del número de mayor magnitud (minuendo):

| | | | | | | |
|---|---|---|---|---|---|---|
| | 8 | 3 | 4 | 1 | (10 | ← minuendo |
| | 1 | 1 | 1 | | | ← acarreo |
| – | 2 | 4 | 6 | 3 | (10 | ← sustraendo |
| | 5 | 8 | 7 | 8 | (10 | ← resultado |

El procedimiento en otras bases es idéntico. Sólo hay que adecuarse a la nueva base y poner atención en restar la magnitud pequeña de la grande:

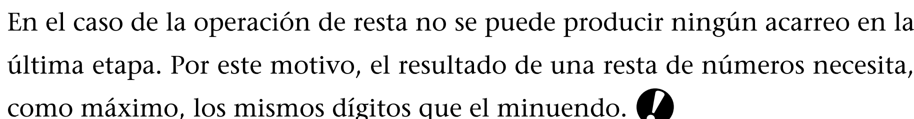
| | | | | | | |
|---|---|---|---|---|---|---|
| | A | F | 1 | 8 | (16 | |
| | | 1 | | | | ← acarreo |
| – | 3 | 5 | 8 | 2 | (16 | |
| | 7 | 9 | 9 | 6 | (16 | ← resultado |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | (2 | |
| | 1 | 1 | 1 | | | | | | | ← acarreo |
| – | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | (2 | |
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | (2 | ← resultado |

**Tabla de resta en base 2**
**acarreo / bit de resta**

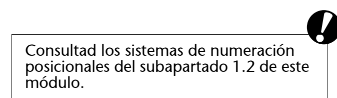
| – | | minuendo 0 | minuendo 1 |
|---|---|---|---|
| sustraendo | 0 | 0/0 | 0/1 |
| sustraendo | 1 | 1/1 | 0/0 |

En el caso de la operación de resta no se puede producir ningún acarreo en la última etapa. Por este motivo, el resultado de una resta de números necesita, como máximo, los mismos dígitos que el minuendo.



## 1.8. Multiplicación y división por potencias de la base de numeración

En un sistema posicional de base fija cada dígito tiene un peso $b^p$ donde $b$ es la base de numeración y $p$ la posición que ocupa el dígito. Los pesos asociados a los dígitos de los números decimales son potencias de 10. Por lo tanto, multiplicar por 10 se traduce en aumentar en una unidad la potencia de 10 asociada a cada dígito y dividir por 10 es equivalente a disminuir en una unidad la potencia de 10 asociada a cada dígito.



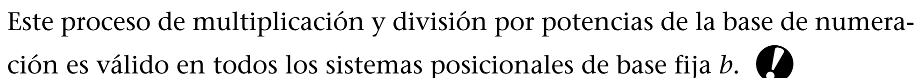
Consultad los sistemas de numeración posicionales del subapartado 1.2 de este módulo.

Según el TFN, el número $56{,}34_{(10} = 5 \cdot 10^1 + 6 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$. Si multiplicamos por 10 tenemos:

$$56{,}34_{(10} \cdot 10 = (5 \cdot 10^1 + 6 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}) \cdot 10 =$$
$$= 5 \cdot 10^2 + 6 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} = 563{,}4_{(10}.$$

El efecto que se obtiene es el desplazamiento de la coma fraccionaria. Multiplicar por 10 un número en base 10 es equivalente a desplazar la coma fraccionaria una posición a la derecha, mientras que dividirlo por 10 es equivalente a desplazar la coma una posición a la izquierda. El proceso se puede extender a la multiplicación y división por una potencia de 10: multiplicar por $10^k$ un número en base 10 equivale a desplazar la coma fraccionaria $k$ posiciones a la derecha, y dividirlo por $10^k$ equivale a desplazar la coma fraccionaria $k$ posiciones a la izquierda.

Este proceso de multiplicación y división por potencias de la base de numeración es válido en todos los sistemas posicionales de base fija *b*.



**Multiplicar por $b^k$** un número en un sistema posicional de base fija *b* equivale a desplazar la coma fraccionaria *k* posiciones a la derecha.

**Dividir por $b^k$** un número en un sistema posicional de base fija *b* equivale a desplazar la coma fraccionaria *k* posiciones a la izquierda.

**Números sin parte fraccionaria**

En un número sin parte fraccionaria desplazar la coma *k* posiciones a la derecha equivale a añadir *k* ceros a la derecha, dado que la parte fraccionaria es cero.

**Multiplicación por una potencia de 2 en binario**

El resultado de multiplicar el $11010_{(2}$ por $2^4$ se consigue si desplazamos la coma fraccionaria 4 posiciones a la derecha:

$$11010_{(2} \cdot 2^4 = 110100000_{(2}$$

Este resultado que obtenemos de forma directa se puede justificar con los cálculos siguientes:

$$11010_{(2} \cdot 2^4 = (1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) \cdot 2^4 =$$
$$= (1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4) = 110100000_{(2}$$

Por lo tanto, $11010_{(2} \cdot 2^4 = 110100000_{(2}$.

**División por una potencia de 2 en binario**

El resultado de dividir el $11100_{(2}$ por $2^2$ se consigue si desplazamos la coma fraccionaria 2 posiciones a la izquierda:

$$11100_{(2} / 2^2 = 111_{(2}$$

Este resultado que obtenemos de forma directa se puede justificar con los cálculos siguientes:

$$11100_{(2} / 2^2 = (1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) / 2^2 =$$
$$= (1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2}) = 111_{(2}$$

Por lo tanto, $11100_{(2} / 2^2 = 111_{(2}$.

La división por una potencia de la base de numeración de un número sin parte fraccionaria puede dar como resultado un número con parte fraccionaria: $11100_{(2} / 2^4 = 1{,}11_{(2}$. Ahora bien, podemos dar el resultado en forma de dos números enteros que reciben el nombre de cociente y resto, donde el cociente tiene relación directa con la parte entera del resultado y el resto con la parte fraccionaria. En este caso, la operación recibe el nombre de división **entera**, mientras que, por oposición, la primera recibe el nombre de división **real**.

El cociente y el resto de la división entera de $11100_{(2}$ por $2^4$ se pueden obtener a partir del resultado de la división real $11100_{(2} / 2^4 = 1{,}11_{(2}$, identificando el cociente con la parte entera (en este caso, el cociente es $1_{(2)}$) y el resto con la parte fraccionaria multiplicada por el divisor (en este caso, el resto es $0{,}11_{(2} \cdot 2^4 = 1100_{(2)}$).

El **cociente** y el **resto** de una división entera de un número entero por una potencia de la base de numeración se pueden obtener a partir del resultado de división real, identificando el cociente con la parte entera y el resto con la parte fraccionaria multiplicada por el divisor.

### Actividades

**10.** Calculad las operaciones siguientes en la base especificada:
**a)** $111011010_{(2} + 100110100_{(2}$
**b)** $2345_{(8} + 321_{(8}$
**c)** $A23F_{(16} + 54A3_{(16}$
**d)** $111011010_{(2} - 100110100_{(2}$
**e)** $2345_{(8} - 321_{(8}$
**f)** $A23F_{(16} - 54A3_{(16}$

**11.** Calculad las operaciones siguientes en la base especificada:
**a)** $62{,}48_{(16} + 35{,}DF_{(16}$
**b)** $111101101{,}11011_{(2} + 100110100{,}111_{(2}$
**c)** $62{,}48_{(16} - 35{,}DF_{(16}$
**d)** $111101101{,}11011_{(2} - 100110100{,}111_{(2}$

**12.** Calculad las multiplicaciones siguientes:
**a)** $128{,}7_{(10} \cdot 10^4$
**b)** $AFD_{(16} \cdot 16^2$
**c)** $1101{,}01_{(2} \cdot 2^2$

**13.** Hallad el cociente y el residuo de las divisiones enteras siguientes:
**a)** $52978_{(10} / 10^3$
**b)** $3456_{(16} / 16^2$
**c)** $100101001001_{(2} / 2^8$

# 2. Representación de los números en un computador

En esta segunda sección se describen sistemas para representar números que se usan para codificar información numérica dentro de los computadores.

**Terminología**

A lo largo del texto, utilizaremos indistintamente **representación** y **codificación** para referirnos a la secuencia de dígitos asociada a un valor numérico en un sistema de representación numérica.

## 2.1. Condicionantes físicos

A pesar de las continuas mejoras tecnológicas, la capacidad de almacenamiento de los computadores es finita. Esto condiciona la representación numérica dentro de los computadores, sobre todo en números con una parte fraccionaria infinita, como por ejemplo los casos muy conocidos de los números $\pi$ o $e$ y, en general, en la representación de números irracionales como $\sqrt{2}$.

Estas limitaciones son parecidas a las que encontramos cuando trabajamos con lápiz y papel. En los cálculos hechos a mano usamos el $3{,}14_{(10}$ o el $3{,}1416_{(10}$ como aproximación a $\pi$. Dentro de los computadores también se trabaja con aproximaciones de los números que no se pueden representar de manera exacta.

Cuando un número no se puede representar de manera exacta dentro de un computador, se comete un **error de representación**. Este error es la distancia entre el número que queremos representar y el número representado realmente.

**Números representables**

Los números que se pueden representar de forma exacta reciben el nombre de *números representables.*

Si representamos el número $\pi$ por el valor $3{,}14_{(10}$, estamos cometiendo un error igual a $|\pi - 3{,}14_{(10}| = 0{,}00159...$, mientras que si trabajamos con el valor $3{,}1416_{(10}$ el error es $|\pi - 3{,}1416_{(10}| = 7{,}3464102...\cdot 10^{-6}$.

Cuando escribimos números lo hacemos de la forma más práctica y adecuada en cada caso. Podemos escribir 03, 3,00, 3,000 o simplemente 3. En cambio, dentro de los computadores hay que seguir una pauta más rígida, un **formato** que especifique y fije el número de dígitos enteros y fraccionarios con que se trabaja. Si suponemos un formato de representación de la forma $x_2x_1x_0,x_{-1}x_{-2}$, donde cada $x_i$ es un dígito binario, el número $3_{(10}$ se tiene que representar como $011{,}00_{(2}$.

Un **formato** de representación numérica es la manera específica en que se tienen que representar los valores numéricos con que se trabaja. El formato fija el conjunto de números que se pueden representar.

En los subapartados siguientes se describen los parámetros que nos ayudan a medir la eficiencia de un formato de representación numérica: el rango de representación, la precisión y el error de representación.

### 2.1.1. Rango de representación

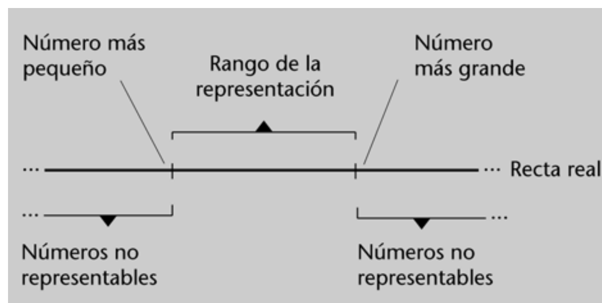
Fijado el formato $x_1x_0,x_{-1}x_{-2}$ en la base 10, sólo podemos representar números entre el $00{,}00_{(10}$ (el número más pequeño representable en este formato) y el $99{,}99_{(10}$ (el número más grande representable en este formato). El número $935_{(10}$ no se puede representar en este formato puesto que no está dentro del intervalo de representación. Los números que se pueden representar en un formato están delimitados dentro de un **intervalo** que recibe el nombre de **rango**.

**Atención**

En un formato sólo se pueden representar un conjunto de números. En un formato con rango [0, 99,99] el número $34{,}789_{(10}$ no se puede representar de forma exacta, porque tiene 3 dígitos fraccionarios.

El **rango** de un formato de representación numérica es el intervalo más pequeño que contiene todos los números representables. Los límites del intervalo son determinados por el número más grande y el número más pequeño que se pueden representar.

La notación que se usa para definir un rango es [*a, b*] donde *a* y *b* son los límites del intervalo en decimal, y forman parte de él.




Los números que están fuera del rango de representación de un formato no son representables en ese formato.

Hay una limitación inherente al número de bits disponibles en un formato de representación: con *n* dígitos en base *b*, disponemos de $b^n$ códigos o combinaciones de dígitos. Cada una de estas combinaciones puede representar un valor numérico. Por lo tanto, con *n* dígitos en base *b* podremos representar un máximo de $b^n$ números diferentes.

Con 5 bits disponemos de $2^5 = 32$ combinaciones diferentes. Podremos representar 32 números. La codificación que se use determinará cuáles son estos números. En base 10 y 4 dígitos disponemos de $10^4$ códigos diferentes (combinaciones de los 4 dígitos decimales). Si empleamos estos códigos para representar números naturales, podremos representar desde el 0000 (0), hasta el 9999 ($10^4 - 1$).

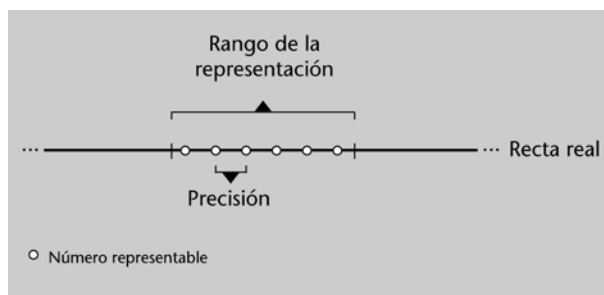
### 2.1.2. Precisión

Estamos habituados a trabajar de manera dinámica con la precisión y la ajustamos automáticamente a nuestras necesidades. Para medir la longitud de una mesa en metros, por ejemplo, trabajamos con dos dígitos fraccionarios. Un formato de estas características nos permitirá distinguir 1,52 m de 1,53 m, pero no de 1,5234 m. Decimos que la precisión de este formato es de 0,01 m, que es la distancia entre dos valores consecutivos representables en este formato.

La **precisión** de un formato de representación numérica es la distancia entre dos números representables consecutivos.

**Nota**

En la mayoría de formatos de representación la distancia entre dos números representables consecutivos cualesquiera es la misma.




### 2.1.3. Error de representación

En un formato de 4 dígitos decimales, de los cuales 2 son fraccionarios, los números son de la forma $x_1x_0{,}x_{-1}x_{-2}$, donde $x_i$ es un dígito decimal. Podemos representar el $12{,}34_{(10}$ o el $45{,}20_{(10}$ de manera exacta, pero no el $15{,}026_{(10}$. Si tenemos que trabajar con este número tendremos que usar una **aproximación**. Podemos aproximarlo por un número representable cercano como el $15{,}03_{(10}$. Trabajar con una aproximación comporta cometer un error. En este caso, el error que se comete es $15{,}03_{(10} - 15{,}026_{(10} = 0{,}004_{(10}$.

**Rangos de representación**

En el formato $x_1x_0{,}x_{-1}x_{-2}$ en base 10, el rango de representación es [0, 99,99]. Un número como el $128_{(10}$, que está fuera del rango de representación, no es representable. No se considera que $99{,}99_{(10}$ sea una aproximación válida para este número en este formato.

El **error de representación** $\varepsilon$ es la distancia entre el número $X$ que queremos representar y el número representable $\widehat{X}$ con el que lo aproximamos. Es decir, $\varepsilon = \left|X - \widehat{X}\right|$.

Los números que no están dentro del rango de representación del formato no son representables ni aproximables. ❗

### 2.1.4. Aproximaciones: truncamiento y redondeo

En el formato $x_1x_0{,}x_{-1}x_{-2}$ en base 10, tanto el $23{,}45_{(10}$ como el $23{,}46_{(10}$ son aproximaciones válidas del $23{,}457_{(10}$. Tenemos que elegir una de las dos po-

sibilidades, por lo cual estableceremos un criterio de elección. Este proceso de elección se denomina **aproximación** o **cuantificación**. Los criterios de elección más habituales son el **truncamiento** y el **redondeo**.

### 1) Truncamiento

El truncamiento es el criterio de cuantificación más directo y sencillo de aplicar, puesto que no comporta ningún tipo de cálculo y consiste en ignorar los dígitos que sobran. En el formato $x_1x_0,x_{-1}x_{-2}$ en base 10 este criterio aproxima el número $23{,}457_{(10}$ por el $23{,}45_{(10}$, fruto de ignorar el último dígito, que no cabe en el formato.

> La cuantificación o aproximación por **truncamiento** consiste en despreciar los dígitos fraccionarios que no caben en el formato. El proceso de truncamiento no comporta ningún tipo de cálculo.

**Truncamiento**

La gran ventaja del truncamiento es que no comporta ningún tipo de cálculo aritmético.

Por truncamiento en el formato $x_1x_0,x_{-1}x_{-2}$ en base 10, tanto el $23{,}451_{(10}$, el $23{,}456_{(10}$ como el $23{,}459_{(10}$ se aproximan por el $23{,}45_{(10}$. Ahora bien, el error cometido en cada caso es diferente. El error es 0,001 para el $23{,}451_{(10}$ (puesto que $|23{,}451_{(10} - 23{,}45_{(10}| = 0{,}001$), 0,006 para el $23{,}456_{(10}$ y 0,009 para el $23{,}459_{(10}$. En todos los casos el error de representación es inferior a la precisión, que es 0,01.

> En una aproximación por truncamiento el **error máximo** de representación es igual a la precisión del formato de representación.

### 2) Redondeo

El $23{,}459_{(10}$ en el formato $x_1x_0,x_{-1}x_{-2}$ en base 10, se aproxima por truncamiento por el $23{,}45_{(10}$ y el error es 0,009. Ahora bien, si aproximáramos el $23{,}459_{(10}$ por el $23{,}46_{(10}$, el error sería 0,001 ($|23{,}46_{(10} - 23{,}459_{(10}| = 0{,}001$), es decir, un error más pequeño. El $23{,}46_{(10}$ está más cerca y sería más exacto trabajar con él. Esta aproximación recibe el nombre de **redondeo** o *aproximación* **al más próximo**.

**Nota**

El número que se obtiene por truncamiento coincide con el que se obtiene por redondeo, siempre que el número resultante por truncamiento sea el número representable más cercano al número que queremos representar.

> La cuantificación o aproximación por **redondeo** consiste en escoger el número representable más cercano al número que queremos representar. El proceso de redondeo comporta operaciones aritméticas.

Si aplicamos el criterio de redondeo en el formato $x_1x_0,x_{-1}x_{-2}$ en base 10, el $23{,}451_{(10}$ se aproxima por el $23{,}45_{(10}$, mientras que el $23{,}456_{(10}$ o el $23{,}459_{(10}$ se aproximan por el $23{,}46_{(10}$ que les es más cercano. El error es 0,001 para el

23,451$_{(10}$, 0,004 para el 23,456$_{(10}$ y 0,001 para el 23,459$_{(10}$. El error cometido es inferior a la mitad de la precisión, es decir, inferior a 0,005 en este caso.

> En una aproximación por redondeo el **error máximo** de representación es igual a la mitad de la precisión del formato de representación.

Una manera sencilla de aplicar el redondeo al número 23,459$_{(10}$ en el formato $x_1x_0,x_{-1}x_{-2}$ en base 10 es sumarle la mitad de la precisión (es decir, 0,005) y a continuación hacer el truncamiento del resultado: 23,459$_{(10}$ + 0,005$_{(10}$ = 23,464$_{(10}$, que truncado a dos dígitos fraccionarios es el 23,46$_{(10}$.

> Para aproximar un número por redondeo tenemos que:
>
> **1)** Sumar la mitad de la precisión del formato de representación al número que se quiere aproximar.
> **2)** Truncar el resultado de la suma según el número de dígitos fraccionarios disponibles en el formato de representación.

**Aproximación por redondeo**

Para aproximar por redondeo el número 1,526246$_{(10}$ en el formato $x_1x_0x_{-1}x_{-2}x_{-3}x_{-4}$ en base 10 procederemos de la forma siguiente:

**1)** Sumar la mitad de la precisión del formato de representación al número que se quiere aproximar:

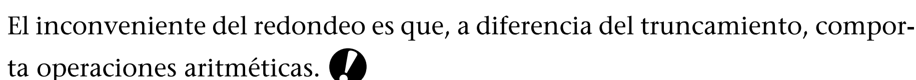
$$1{,}526264_{(10} + 0{,}00005_{(10} = 1{,}526314_{(10}$$

**2)** Truncar el resultado de la suma según el número de dígitos fraccionarios disponibles en el formato de representación:

$$1{,}526314_{(10} \rightarrow 1{,}5263_{(10}$$

Por lo tanto, el número 1,526264$_{(10}$ se aproxima por redondeo en este formato por el 1,5263$_{(10}$.

El inconveniente del redondeo es que, a diferencia del truncamiento, comporta operaciones aritméticas.



### 2.1.5. Desbordamiento

Al hacer operaciones aritméticas con números en un formato determinado, nos podemos encontrar con que el resultado esté fuera del rango de representación. Es lo que se conoce como **desbordamiento**.

> El **desbordamiento** aparece cuando el resultado de una operación supera el rango de representación.

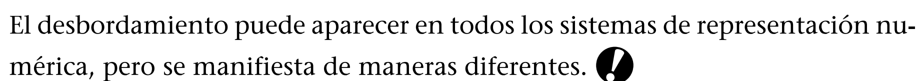
**Terminología**

En inglés, el término *desbordamiento* recibe el nombre de *overflow* .

En un formato de 6 bits, la operación de suma siguiente produce desbordamiento, porque el resultado no cabe en 6 bits:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | 1 | 1 | | | | | ← acarreo |
| | | 1 | 0 | 1 | 1 | 1 | 0 | (2 | |
| + | | 1 | 0 | 0 | 1 | 0 | 1 | (2 | |
| | 1 | **0** | **1** | **0** | **0** | **1** | **1** | (2 | ← resultado |
| | ↑ | | | | | | | | |
| | desbordamiento | | | | | | | | |

El desbordamiento puede aparecer en todos los sistemas de representación numérica, pero se manifiesta de maneras diferentes.



Hay un tipo especial de desbordamiento que recibe el nombre de **desbordamiento a cero** y que aparece cuando un número de magnitud menor que la precisión del formato, pero diferente de cero, finalmente se acaba representando, debido al error de representación, como cero. Esta situación es relevante porque operaciones como la división que, a priori, no tendrían que presentar ninguna dificultad pueden volverse irresolubles.

**Terminología**

En inglés, el desbordamiento a cero recibe el nombre de *underflow* .

### Actividades

**14.** Determinad el rango y la precisión de los formatos de coma fija sin signo $x_1x_0,x_{-1}x_{-2}x_{-3}$ y $x_2x_1x_0,x_{-1}x_{-2}$ donde $x_i$ es un dígito decimal.

**15.** Determinad si el número 925,4 se puede representar en los formatos indicados en la actividad 14.

**16.** Representad en el formato de coma fija y sin signo $x_1x_0,x_{-1}x_{-2}$, donde $x_i$ es un dígito decimal, los números siguientes:
**a)** $10_{(10}$
**b)** $10{,}02_{(10}$
**c)** $03{,}1_{(10}$
**d)** $03{,}2_{(10}$

**17.** Determinad la cantidad de números que se pueden representar en el formato $x_2x_1x_0,x_{-1}x_{-2}x_{-3}$, donde $x_i$ es un dígito decimal.

**18.** Calculad el error de representación que se comete cuando representamos en el formato $x_2x_1x_0,x_{-1}x_{-2}$, donde $x_i$ es un dígito decimal, los números siguientes:
**a)** $223{,}45_{(10}$
**b)** $45{,}89_{(10}$
**c)** $55{,}6356_{(10}$
**d)** $23{,}56_{(10}$

**19.** Escoged el formato hexadecimal que use el mínimo número de dígitos y que permita representar el número $16{,}25_{(10}$ de manera exacta. ¿Cuál es el rango y la precisión del formato?

**20.** ¿Cuál es el número más pequeño que hay que sumar a $8341_{(10}$ para que se produzca desbordamiento en una representación decimal (base 10) de cuatro dígitos?

## 2.2. Números naturales

Los números naturales son los números sin parte fraccionaria y sin signo. Es decir, son los miembros de la sucesión: 0, 1, 2, 3, 4, 5, 6...

Dentro de los computadores los números naturales se representan en base 2, la precisión es 1 (puesto que no hay bits fraccionarios) y el rango depende del número de bits disponibles en el formato.

> El **rango** de representación de los números naturales en un formato de $n$ bits es, en decimal, $[0,2^n - 1]$ y su **precisión** es 1.

El rango de representación se puede ampliar si aumentamos el número de bits de la representación. La ampliación del número de bits de un formato de representación recibe el nombre de **extensión**.

> La **extensión** de los números naturales representados en un formato de $n$ bits a un formato de $m$ bits, con $m > n$, se consigue añadiendo, a la izquierda de la codificación, los ceros necesarios hasta completar los $m$ bits del formato nuevo.

La representación del número natural $10_{(10}$ en un formato de 5 bits es $01010_{(2}$. La extensión de esta codificación a un formato de 8 bits se consigue añadiendo ceros a la izquierda hasta completar los 8 bits del formato nuevo, con lo cual la nueva codificación será $00001010_{(2}$.

**Cambio de base del número $10_{(10}$ a base 2**

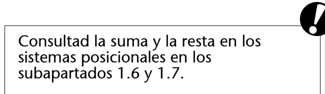
Siguiendo el método basado en el teorema de la división entera:

$10 = 5 \cdot 2 + 0$
$5 = 2 \cdot 2 + 1$
$2 = 1 \cdot 2 + 0$
$1 = 0 \cdot 2 + 1$

$10_{(10} = 1010_{(2}$

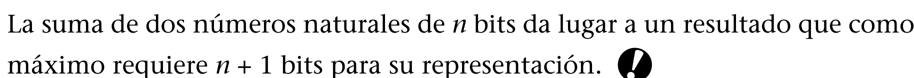
Las operaciones de suma y de resta siguen las pautas expuestas anteriormente. Si se produce un acarreo en la última etapa de suma, hay desbordamiento:

$$
\begin{array}{rcccccccl}
 & 1 & 1 & & & & & & \leftarrow \text{acarreo} \\
 & & 1 & 1 & 1 & 0 & 1 & 0_{(2} & \\
+ & & 0 & 1 & 0 & 1 & 0 & 1_{(2} & \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 1 & 1_{(2} & \leftarrow \text{resultado} \\
 & \uparrow & & & & & & & \\
 & \text{desbordamiento} & & & & & & &
\end{array}
$$



Consultad la suma y la resta en los sistemas posicionales en los subapartados 1.6 y 1.7.

La suma de dos números naturales de $n$ bits da lugar a un resultado que como máximo requiere $n + 1$ bits para su representación.

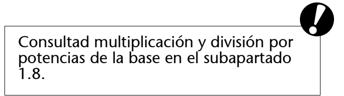


**Atención**

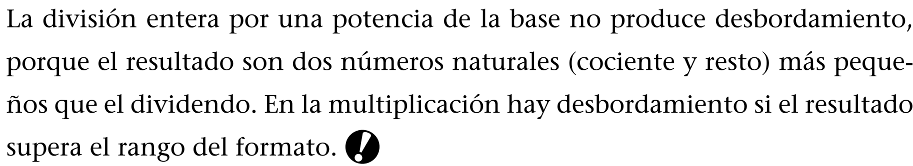
La resta de dos naturales no puede producir desbordamiento porque restamos la magnitud pequeña de la grande. Restar la magnitud grande de la pequeña no es una operación válida dentro de los naturales, porque el resultado sería un número con signo.

> El **desbordamiento** en la suma de dos números naturales se produce cuando tenemos un acarreo en la última etapa de suma. La operación de resta de números naturales no puede dar lugar a desbordamiento.

Las operaciones de multiplicación y división entera por potencias de la base de numeración se ajustan a los procedimientos ya descritos.



Consultad multiplicación y división por potencias de la base en el subapartado 1.8.



La división entera por una potencia de la base no produce desbordamiento, porque el resultado son dos números naturales (cociente y resto) más pequeños que el dividendo. En la multiplicación hay desbordamiento si el resultado supera el rango del formato.

**La división de dos naturales**

La operación de división sobre números naturales debe ser la división entera dado que los números naturales no tienen parte fraccionaria.

### 2.3. Números enteros

Los enteros son los números con signo y sin parte fraccionaria, incluyendo el cero: ... –3, –2, –1, 0, +1, +2, +3 ... Se diferencian de los naturales por la presencia de un signo que indica si la magnitud es positiva o negativa. Este signo se puede incorporar a la codificación de los números dentro de los computadores de varias maneras. En los apartados siguientes describimos las más utilizadas en los computadores: signo y magnitud, y complemento a 2.

#### 2.3.1. Representación de enteros en signo y magnitud en base 2

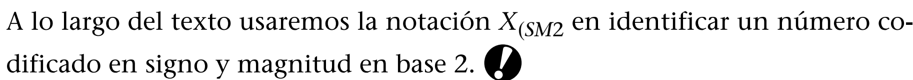
Signo y magnitud es, probablemente, la forma más intuitiva de representar números con signo. En **signo y magnitud**, el bit más significativo (MSB) almacena el signo y el resto codifica la magnitud. Un 1 en el dígito más significativo indica signo negativo, mientras que un 0 indica signo positivo.

**MSB y LSB**

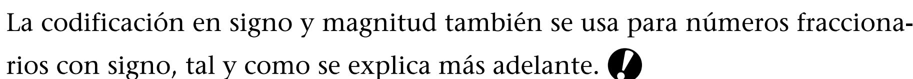
MSB es la abreviación de *most significant bit*, es decir, el bit más significativo de la representación, que se corresponde con el dígito del extremo izquierdo. LSB es la abreviación de *least significant bit*, es decir, el bit menos significativo de la representación, que se corresponde con el dígito del extremo derecho.

Así, si la cadena de bits 101001 es un número en signo y magnitud, sabremos que es un número negativo, porque el bit más significativo es 1; y que la magnitud es $01001_{(2}$, que en base 10 es el $9_{(10}$. Esta cadena de bits codifica el $-9_{(10}$.

> Un número codificado en **signo y magnitud** con $n$ bits viene dado por la cadena de bits $x_{n-1}x_{n-2}\cdots x_1x_0$, donde $x_{n-1}$ codifica el signo y $x_{n-2}\cdots x_1x_0$, la magnitud. El signo es positivo si $x_{n-1}$ es 0, y negativo si es 1.



A lo largo del texto usaremos la notación $X_{(SM2}$ en identificar un número codificado en signo y magnitud en base 2.



La codificación en signo y magnitud también se usa para números fraccionarios con signo, tal y como se explica más adelante.

**Representación en signo y magnitud**

Para representar el $-12_{(10}$ en signo y magnitud, 6 dígitos y base 2, tenemos que pasar la magnitud $12_{(10}$ a base 2 ($12_{(10} = 1100_{(2}$) y poner el bit más significativo de la representación (el bit de más a la izquierda) a 1. La representación con 6 bits es, pues, $101100_{(SM2}$.

El $+12_{(10}$ se representa en el mismo formato como $001100_{(SM2}$. Sólo cambia el bit más significativo, porque la magnitud es la misma.

**Cambio de base del número $12_{(10}$ a base 2**

Aplicando la división entera:

$12 = 6 \cdot 2 + 0$
$6 = 3 \cdot 2 + 0$
$3 = 1 \cdot 2 + 1$
$1 = 0 \cdot 2 + 1$

$12_{(10} = 1100_{(2}$

### Rango de representación en signo y magnitud y base 2

El formato de signo y magnitud es simétrico, es decir, se pueden representar tantos valores positivos como negativos. Con 4 bits y signo y magnitud tendremos 1 bit (el más significativo) para el signo y 3 para la magnitud:

- Valores posibles de signo:

$$0 \rightarrow +$$
$$1 \rightarrow -$$

- Valores posibles para la magnitud:

| | |
|---|---|
| $000_{(2} = 0_{(10}$ | $100_{(2} = 4_{(10}$ |
| $001_{(2} = 1_{(10}$ | $101_{(2} = 5_{(10}$ |
| $010_{(2} = 2_{(10}$ | $110_{(2} = 6_{(10}$ |
| $011_{(2} = 3_{(10}$ | $111_{(2} = 7_{(10}$ |

**Ventajas del formato de signo y magnitud**

El formato de signo y magnitud tiene ventajas a la hora de hacer multiplicaciones: se operan por separado las magnitudes y los signos y, posteriormente, se juntan los resultados obtenidos de manera independiente.

Combinando signo y magnitud podemos representar los valores enteros entre –7 y +7. Por lo tanto, el rango de representación es [ –7 , +7].

En general, en signo y magnitud en base 2, el **rango** de enteros representable con *n* bits es, en decimal,

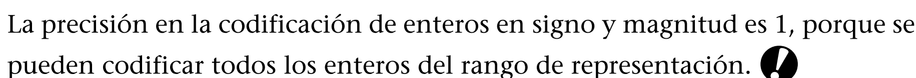
$$[-(2^{n-1} - 1), 2^{n-1} - 1]$$

Si aplicamos esta expresión al caso de 4 bits, tenemos:

$$[-(2^{4-1} - 1), 2^{4-1} - 1] = [-(2^3 - 1), 2^3 - 1] = [-7, +7]$$

que es el rango al que habíamos llegado de manera experimental.

La precisión en la codificación de enteros en signo y magnitud es 1, porque se pueden codificar todos los enteros del rango de representación.



Si fuera necesario ampliar el rango de representación tendríamos que hacer una extensión del formato de signo y magnitud.

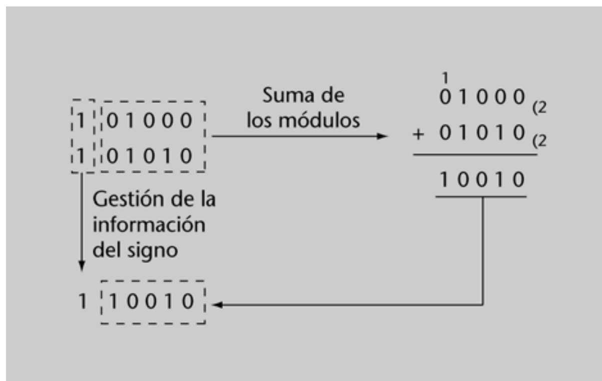
La **extensión** de *n* a *m* bits, con $m > n$, de los números en signo y magnitud se consigue añadiendo, a la izquierda de la magnitud, los ceros necesarios para completar los *m* bits, manteniendo el bit del extremo izquierdo para la codificación del signo.

Por consiguiente, el entero negativo $11010_{(SM2}$ codificado en signo y magnitud y 5 bits se puede extender a un formato de 8 bits añadiendo ceros a la derecha del signo, de forma que la codificación de este mismo número en el nuevo formato sería **1000**$1010_{(SM2}$. La extensión de números positivos se hace del mismo modo. La extensión a 8 bits de la codificación en signo y magnitud del entero positivo $01010_{(SM2}$ nos lleva al **0000**$1010_{(SM2}$.

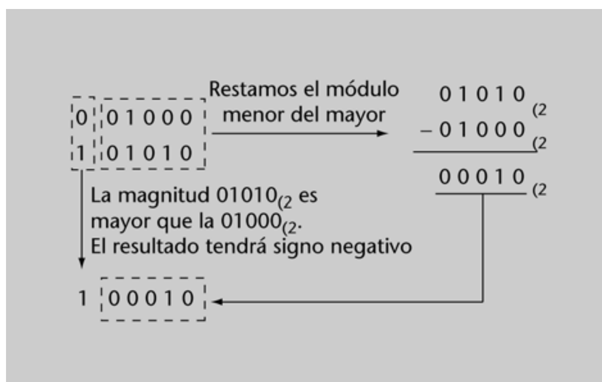
### 2.3.2. Suma y resta en signo y magnitud

La suma de dos números positivos o dos negativos en signo y magnitud es sencilla. Tenemos que hacer la suma de las magnitudes y dar al resultado el signo de los operandos. La suma de las magnitudes puede producir desbordamiento.

La suma de los números $101000_{(SM2}$ y $101010_{(SM2}$ codificados en signo y magnitud y 6 bits, es la siguiente:




La suma de un positivo y un negativo es más compleja: hay que analizar las magnitudes para saber cuál es la mayor, restar la magnitud pequeña de la grande y aplicar al resultado el signo de la magnitud mayor. El procedimiento de suma de los números $001000_{(SM2}$ y $101010_{(SM2}$ codificados en signo y magnitud y 6 bits, es el siguiente:




La suma de dos números de mismo signo y la resta de números de signo contrario puede producir desbordamiento.

> En signo y magnitud, hay **desbordamiento** en la suma de dos números del mismo signo o en la resta de números de signo contrario cuando aparece un acarreo en la última etapa de suma o resta de las magnitudes.
>
> Ni la suma de un positivo y un negativo, ni la resta de números del mismo signo pueden producir desbordamiento.

En la suma de $101010_{(SM2}$ y el $111010_{(SM2}$, en signo y magnitud y 6 bits, examinamos, en primer lugar, los signos. Son dos números negativos, puesto que el bit de mayor peso de ambos es 1. Por lo tanto, procederemos a la suma de las magnitudes:

```
   1 | 1   1          ← acarreo
     |
     | 0 1 0 1 0 (2
 +   | 1 1 0 1 0 (2
 ----+--------------
   1 | 0 0 1 0 0 (2   ← resultado
   ↑ |
     |
 desbordamiento
```

La suma de las magnitudes produce desbordamiento, puesto que tenemos un acarreo en la última etapa. Por lo tanto, el resultado no cabe en el formato definido y no se puede representar.

Los inconvenientes principales del sistema de signo y magnitud son la complejidad de las operaciones de suma y resta y la existencia de dos representaciones para el 0: un "0 positivo", cuando la magnitud es 0 y el signo también; y un "0 negativo", cuando la magnitud es 0 y el signo 1.

### 2.3.3. Representación en complemento a 2

El **complemento a 2**, abreviado habitualmente por **Ca2** o **C2**, es un sistema de representación de números con signo en base 2. Actualmente, el Ca2 es el sistema más empleado para codificar números enteros en los computadores porque presenta dos ventajas: una codificación única para el cero, y simplicidad en las operaciones de suma y resta.

> Los **números positivos en Ca2** se codifican de la misma forma que en signo y magnitud: el bit del extremo izquierdo es 0, para indicar signo positivo, y el resto contiene la magnitud.
>
> La codificación de un **número negativo** $-X$ en Ca2 es el resultado en binario de la operación $2^n - |X|$, donde $|X|$ es el valor absoluto de $X$.

**Inconvenientes del formato de Ca2**

Sin que afecte a la eficiencia de los computadores, los valores de las magnitudes negativas codificadas en Ca2 son más difíciles de reconocer para nosotros.

A lo largo del texto utilizaremos la notación $X_{(Ca2}$ para identificar un número codificado en complemento a 2.

**Codificación de números negativos en Ca2**

Para hallar la codificación en Ca2 y 6 bits del valor $-11010_{(2}$, hacemos la operación siguiente:

$$2^6 - |X| = 1000000_{(2} - 11010_{(2} = 100110_{(Ca2}$$

$$\begin{array}{cc|cccccc l} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & _{(2} \\ & _1 & _1 & _1 & _1 & _1 & & & \\ - & & & 1 & 1 & 0 & 1 & 0 & _{(2} \\ \hline & 0 & 1 & 0 & 0 & 1 & 1 & 0 & _{(Ca2} \end{array}$$

Así pues, la codificación en Ca2 y 6 bits del valor $-11010_{(2}$ es $100110_{(Ca2}$.

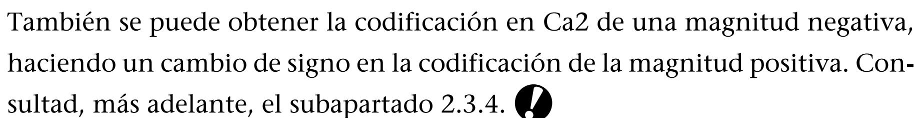
La codificación del valor $+11010_{(2}$ en Ca2 coincide con la codificación en signo y magnitud. Tendremos un 0 para el signo y a continuación 5 bits con la magnitud: el valor $+11010_{(2}$ se codifica en Ca2 como $011010_{(Ca2}$.

Para hallar la codificación en Ca2 y 8 bits del valor $-11010_{(2}$, hacemos la operación siguiente:

$$2^8 - |X| = 100000000_{(2} - 11010_{(2} = 11100110_{(Ca2}$$

$$\begin{array}{cc|cccccccc l} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & _{(2} \\ & _1 & _1 & _1 & _1 & _1 & _1 & _1 & & & \\ - & & & & & 1 & 1 & 0 & 1 & 0 & _{(2} \\ \hline & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & _{(Ca2} \end{array}$$

Así pues, la codificación en Ca2 y 8 bits del valor $-11010_{(2}$ es $11100110_{(Ca2}$.



También se puede obtener la codificación en Ca2 de una magnitud negativa, haciendo un cambio de signo en la codificación de la magnitud positiva. Consultad, más adelante, el subapartado 2.3.4.

### Rango de representación en Ca2

La tabla siguiente muestra los enteros representables con 4 bits en signo y magnitud y en complemento a 2 y su correspondencia:

| Decimal | Signo y magnitud | Ca2 |
|---|---|---|
| +7 | 0111 | 0111 |
| +6 | 0110 | 0110 |
| +5 | 0101 | 0101 |
| +4 | 0100 | 0100 |
| +3 | 0011 | 0011 |
| +2 | 0010 | 0010 |
| +1 | 0001 | 0001 |
| 0 | 0000<br>1000 | 0000 |
| –1 | 1001 | 1111 |
| –2 | 1010 | 1110 |

**Tabla**

En la tabla vemos que los positivos se codifican igual en Ca2 y signo y magnitud. El rango de los positivos es el mismo en los dos sistemas. En cambio, en Ca2 tenemos un negativo más que en signo y magnitud. Esto es debido a que en Ca2 hay una representación única del cero, mientras que en signo y magnitud tiene dos.

| Decimal | Signo y magnitud | Ca2 |
|---|---|---|
| –3 | 1011 | 1101 |
| –4 | 1100 | 1100 |
| –5 | 1101 | 1011 |
| –6 | 1110 | 1010 |
| –7 | 1111 | 1001 |
| –8 | no es representable | 1000 |

En Ca2 y 4 bits se puede representar desde el $-8_{(10}$ hasta el $+7_{(10}$.

En general, el **rango** de enteros representables con *n* bits en Ca2 es, en decimal:

$$[-2^{n-1}, 2^{n-1} - 1]$$

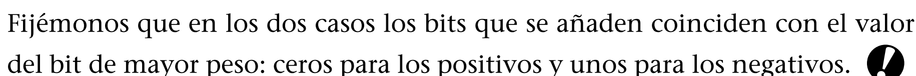
Con 4 bits, el rango es: $[-2^{4-1}, 2^{4-1} - 1] = [-2^3, 2^3 - 1] = [-8, +7]$

En Ca2, para aumentar el número de bits con que se codifica un entero positivo se puede seguir el mismo procedimiento que en signo y magnitud. En cambio, la extensión para los enteros negativos es diferente. El $-10_{(10}$ en Ca2 y 5 bits es el $10110_{(Ca2}$, mientras que con 8 bits se codifica como $11110110_{(Ca2}$. La diferencia entre las codificaciones es que en la segunda se han añadido tres 1 a la izquierda.

**Ejemplo**

En Ca2, el $-10_{(10}$ se codifica con 5 bits, por el 10110:
$2^5 - |-10_{(10}| = 32_{(10} - 10_{(10} =$
$= 22_{(10} = 10110_{(Ca2}$
En Ca2, el $-10_{(10}$ se codifica con 8 bits, por el 11110110:
$2^8 - |-10_{(10}| = 256_{(10} - 10_{(10} =$
$= 246_{(10} = 11110110_{(Ca2}$

Fijémonos que en los dos casos los bits que se añaden coinciden con el valor del bit de mayor peso: ceros para los positivos y unos para los negativos.



En Ca2, **para extender** un formato de *n* bits a *m* bits, con $m > n$, se copia a la izquierda el bit de más peso las veces necesarias para completar los *m* bits. Este proceso recibe el nombre de **extensión del signo**.

**En Ca2 el bit de mayor peso indica el signo**

En Ca2, un 1 en el bit de mayor peso indica que el número es negativo, mientras que un 0 indica que es positivo.

### 2.3.4. Cambio de signo en complemento 2

Haremos un cambio de signo de un número en Ca2, si seguimos los pasos siguientes:

1) Hacer el complemento bit a bit de la codificación en Ca2.
2) Sumar 1 al bit menos significativo de la codificación.

**Complemento bit a bit**

Se entiende por *complemento bit a bit*, la sustitución de cada bit por su complementario.

En base 2, el complementario del 0 es el 1 y el del 1 es el 0.

**Cambio de signo en complemento a 2**

Para hacer el cambio de signo del valor numérico $11000110_{(2}$ (que si seguimos el procedimiento explicado en el apartado siguiente veríamos que se trata del $-58_{(10)}$), que está codificado en complemento a 2 y 8 dígitos, hacemos la operación siguiente:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | (Ca2 | ← valor numérico inicial |
| | | | | | | | 1 | | | ← acarreo |
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | (Ca2 | ← complemento bit a bit de la expresión inicial |
| + | | | | | | | | 1 | | ← sumamos 1 al bit menos significativo del formato |
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (Ca2 | |

De esta operación resulta el $00111010_{(Ca2}$, que codifica la misma magnitud, pero con signo positivo.

El cambio de signo de un número en Ca2 también se puede conseguir si examinamos los bits de derecha a izquierda y:

**1)** Mantenemos los mismos bits hasta encontrar el primer 1 (incluyéndolo).
**2)** Hacemos el complemento bit a bit del resto.

**Cambio de signo en complemento a 2**

Para hacer el cambio de signo del 11000110 que está en Ca2 y 8 bits, lo examinamos de derecha a izquierda, haciendo el complementando bit a bit después del primer 1:

```
11000110
      ^ se mantienen los bits hasta aquí (primer 1 que encontramos incluido)
      10
     ^ se complementan los bits a partir de este punto
001110
```

De esta operación resulta el 00111010, que codifica la misma magnitud pero con signo positivo.

El cambio de signo del 00011110, que está en Ca2 y 8 bits, se obtiene siguiendo el mismo procedimiento:

```
00011110
      ^ se mantienen los bits hasta aquí (primer 1 que encontramos incluido)
      10
     ^ se complementan los bits a partir de este punto
111000
```

El resultado es 11100010, que codifica la misma magnitud pero con signo negativo.

El cambio de signo se puede usar como alternativa a la operación $2^n - |X|$ para encontrar la codificación en Ca2 de una magnitud negativa.

> La codificación en Ca2 de una magnitud negativa se puede obtener aplicando un cambio de signo a la codificación en Ca2 de la magnitud positiva.

La operación es reversible. Si aplicamos un cambio de signo a la codificación en Ca2 de una magnitud negativa, encontraremos la codificación de la positiva.

### 2.3.5. Magnitud de los números en complemento a 2

Como en signo y magnitud, la magnitud decimal de un número positivo codificado en Ca2 se puede conocer aplicando el TFN:

$$0101_{(2} = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = +5_{(10}$$

En Ca2, la cadena de bits 0101 codifica el valor $+5_{(10}$.

Para encontrar la magnitud decimal de un número negativo en Ca2, disponemos de dos alternativas:

**1)** Aplicando el TFN, como en el caso de los positivos, pero considerando que el bit de mayor peso es negativo.

**Magnitud decimal de un valor negativo codificado en Ca2 aplicando el TFN**

Aplicamos el TFN para encontrar la magnitud decimal que codifica la cadena de bits 10001010 en Ca2, considerando que el primer bit es negativo:

$$10001010_{(Ca2} = -1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -128 + 10 = -118_{(10}$$

El 10001010 en Ca2 codifica el valor decimal –118.

**2)** Aplicar un cambio de signo a la representación en Ca2 del valor negativo y encontrar la magnitud positiva.

**Magnitud decimal de un valor negativo en Ca2 por cambio de signo**

Para conocer la magnitud decimal que codifica el 10001010 en Ca2:

**1)** Aplicamos un cambio de signo:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | (Ca2 | ← Valor numérico inicial |
| | | | | | | | 1 | | | ← Acarreo |
| | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | (Ca2 | ← Complemento bit a bit de la expresión inicial |
| + | | | | | | | | 1 | | ← Sumamos 1 al bit menos significativo del formato |
| | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | (Ca2 | |

**2)** Aplicamos el TFN al resultado:

$$01110110_{(Ca2} = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = +118_{(10}$$

Por lo tanto, la cadena de bits 10001010 codifica en Ca2 el entero decimal –118.

### 2.3.6. Suma en complemento a 2

El mecanismo de suma en Ca2 es el mismo que el utilizado en cualquier otro sistema posicional. Tenemos que saber reconocer, sin embargo, cuándo se produce desbordamiento.

Consultad la suma y la resta en los sistemas posicionales en los subapartados 1.6 y 1.7.

**Suma de dos valores positivos en Ca2**

Consideremos la suma de dos números positivos en Ca2 y 6 bits siguiente:

> Podemos encontrar la correspondencia entre los números en Ca2 y los valores decimales, aplicando cualquiera de los dos métodos expuestos en el apartado 2.3.5.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 1 | 0 | (Ca2 | → | | +10 | (10 |
| + | 0 | 0 | 0 | 1 | 0 | 1 | (Ca2 | → | + | +5 | (10 |
| | 0 | 0 | 1 | 1 | 1 | 1 | (Ca2 | → | | +15 | (10 |

(resultado correcto)

Sabemos que el resultado es correcto porque hemos hecho la suma de dos positivos y obtenemos una magnitud positiva. Cuando el resultado supera el rango de representación, la suma de dos positivos genera una magnitud negativa, como en el caso siguiente:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | | | ← acarreo | | | | |
| | 0 | 1 | 0 | 1 | 1 | 0 | (Ca2 | → | | +22 | (10 |
| + | 0 | 0 | 1 | 1 | 1 | 1 | (Ca2 | → | + | +15 | (10 |
| | 1 | 0 | 0 | 1 | 0 | 1 | (Ca2 | → | | −27 | (10 |

(desbordamiento)

En Ca2 y 6 bits, el rango es $[-2^{6-1}, +2^{6-1} - 1] = [-32, +31]$. El resultado de esta suma tendría que ser $+37_{(10}$ ($22_{(10}+15_{(10} = +37_{(10}$), que queda fuera del rango.

> Hay **desbordamiento** en la suma de dos números positivos codificados en Ca2 cuando el resultado es negativo.

**Suma de dos valores negativos en Ca2**

De manera análoga, el resultado de la suma de dos negativos en Ca2 es correcto cuando se obtiene una magnitud negativa, y erróneo cuando se obtiene una positiva.

Consideremos la suma de dos números negativos en Ca2 y 6 bits siguiente:

> Podemos encontrar la correspondencia entre los números en Ca2 y los valores decimales aplicando cualquiera de los dos métodos expuestos en el apartado 2.3.5.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | | | | | ← acarreo | | | | |
| | | 1 | 1 | 1 | 0 | 1 | 0 | (Ca2 | → | | − 6 | (10 |
| + | | 1 | 1 | 0 | 1 | 0 | 1 | (Ca2 | → | + | −11 | (10 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 1 | (Ca2 | → | | −17 | (10 |

(resultado correcto)

El resultado es correcto, aunque haya un acarreo en la última etapa de suma, porque se obtiene una magnitud negativa en la suma de dos negativos. En cambio, el resultado de la suma siguiente es erróneo:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | 1 | 1 | 1 | | | ← acarreo | | | |
| | | 1 | 0 | 0 | 1 | 1 | 0 | (Ca2 | → | −26 | (10 |
| + | | 1 | 0 | 1 | 1 | 1 | 1 | (Ca2 | → | + −17 | (10 |
| | 1 | 0 | 1 | 0 | 1 | 0 | 1 | (Ca2 | → | +21 | (10 |
| | | | | | | | | (desbordamiento) | | | |

Como en el caso anterior, el acarreo en la última etapa se tiene que despreciar, pero sumando dos negativos hemos obtenido un positivo. Por lo tanto, hay desbordamiento.

> Hay **desbordamiento** en la suma de dos números negativos codificados en Ca2 cuando el resultado es positivo. El acarreo de la última etapa no indica desbordamiento y se tiene que despreciar en todos los casos.

**Suma de un valor positivo y un valor negativo en Ca2**

El resultado de la suma de un positivo y un negativo en Ca2 siempre es correcto.

Podemos encontrar la correspondencia entre los números en Ca2 y los valores decimales, aplicando cualquiera de los dos métodos expuestos en el apartado 2.3.5.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | | | | | ← acarreo | | | |
| | | 1 | 1 | 1 | 0 | 1 | 0 | (Ca2 | → | −6 | (10 |
| + | | 0 | 1 | 0 | 1 | 0 | 1 | (Ca2 | → | + +21 | (10 |
| | 1 | 0 | 0 | 1 | 1 | 1 | 1 | (Ca2 | → | +15 | (10 |
| | | | | | | | | (resultado correcto) | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | (Ca2 → | −26 | (10 |
| + | 0 | 1 | 0 | 0 | 0 | 1 | (Ca2 → | + +17 | (10 |
| | 1 | 1 | 0 | 1 | 1 | 1 | (Ca2 → | −9 | (10 |
| | | | | | | | (resultado correcto) | | |

La aparición de un acarreo en la última etapa, como la operación de la izquierda, no indica desbordamiento y se tiene que despreciar.

> La suma de un número positivo y un número negativo codificados en Ca2 no puede producir desbordamiento. El acarreo que se puede producir en la última etapa no indica desbordamiento y se tiene que despreciar en todos los casos.

### 2.3.7. Resta en complemento a 2

El procedimiento que se sigue para hacer la resta en Ca2 es aplicar un cambio de signo al sustraendo (operando que queremos restar) y hacer una operación de suma. Sumamos el minuendo con el sustraendo al cual hemos cambiado el signo.

La operación de resta en Ca2 se reduce a una operación de suma una vez se ha cambiado el signo del sustraendo.

La resta $011010_{(Ca2} - 001011_{(Ca2}$ ($26_{(10} - 11_{(10)}$) nos servirá de ejemplo para ilustrar el procedimiento:

**1)** Aplicamos un cambio de signo al sustraendo:

**a)** Hacemos el complemento bit a bit de 001011, con el que se obtiene 110100.
**b)** Sumamos 1 al bit menos significativo:

$$\begin{array}{rl} 1\ 1\ 0\ 1\ 0\ 0 & _{(Ca2} \\ +\qquad\quad 1 & _{(Ca2} \\ \hline 1\ 1\ 0\ 1\ 0\ 1 & _{(Ca2} \end{array}$$

**2)** Hacemos la operación de suma:

$$\begin{array}{rl|rl} 1\ \ 1 & \leftarrow \text{acarreo} & & \\ 0\ 1\ 1\ 0\ 1\ 0 & _{(Ca2} & \rightarrow \quad +26 & _{(10} \\ +\ 1\ 1\ 0\ 1\ 0\ 1 & _{(Ca2} & \rightarrow \quad + \ -11 & _{(10} \\ \hline 1\ 0\ 0\ 1\ 1\ 1\ 1 & _{(Ca2} & \rightarrow \quad +15 & _{(10} \end{array}$$

(resultado correcto)

El resultado es correcto. El acarreo de la última etapa se tiene que despreciar y no se produce desbordamiento. Recordamos que la suma de un número positivo y un número negativo no puede dar lugar a desbordamiento.

Podemos encontrar la correspondencia entre los números en Ca2 y los valores decimales, aplicando cualquiera de los dos métodos expuestos en el apartado 2.3.5.

### 2.3.8. Multiplicación por $2^k$ de números en complemento a 2

Como hemos visto, multiplicar por $2^k$ en sistemas de numeración posicionales de base 2 equivale a desplazar la coma fraccionaria *k* posiciones a la derecha. En el caso de los enteros, este efecto se consigue añadiendo a la derecha *k* ceros:

Ved la multiplicación y la división por potencias de la base de numeración en el subapartado 1.8 de este módulo.

$$000101_{(Ca2} \cdot 2^2 = 010100_{(Ca2} \qquad \text{(en decimal, } +5_{(10} \cdot 2^2 = +20_{(10)}\text{)}$$

El procedimiento también es válido para números negativos en Ca2:

$$111011_{(Ca2} \cdot 2^2 = 101100_{(Ca2} \qquad \text{(en decimal, } -8_{(10} \cdot 2^2 = -32_{(10}\text{)}$$

El resultado de **multiplicar por $2^k$ un número en Ca2** se consigue añadiendo *k* ceros a la derecha.

**Por aplicación del TFN**

$000101_{(2} = 2^2 + 2^0 = 5_{(10}$
$010100_{(2} = 2^4 + 2^2 = 20_{(10}$
$111011_{(2} = -2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -5_{(10}$
$101100_{(2} = -2^5 + 2^3 + 2^2 = -20_{(10}$

Una vez hemos fijado un formato de *n* bits, añadir *k* ceros a la derecha nos obliga a perder *k* bits de la izquierda, lo cual puede producir desbordamiento. En Ca2 y 6 bits, las operaciones siguientes producen desbordamiento:

$000101_{(Ca2} \cdot 2^4 = 010000_{(Ca2}$ (en decimal, $+5_{(10} \cdot 2^4 = +16_{(10}$)
$111000_{(Ca2} \cdot 2^4 = 000000_{(Ca2}$ (en decimal, $-8_{(10} \cdot 2^4 = 0_{(10}$ )

Se produce **desbordamiento** al multiplicar un número en Ca2 por $2^k$ cuando cambia el bit de signo o bien si se pierde uno o más bits significativos. Los bits significativos son, para los positivos los 1 y para los negativos los 0.

### Actividades

**21.** Convertid los valores decimales siguientes a binarios en los sistemas de representación de signo y magnitud y complemento a 2, con un formato entero de 8 bits:
**a)** 53
**b)** –25
**c)** 93
**d)** –1
**e)** –127
**f)** –64

**22.** Si tenemos los números binarios 00110110, 11011010, 01110110, 11111111 y 11100100, ¿cuáles son los equivalentes decimales considerando que son valores binarios representados en signo y magnitud?

**23.** Repetid el ejercicio anterior considerando que las cadenas de bits son números en complemento a 2.

**24.** Si tenemos las cadenas de bits siguientes $A$ = 1100100111, $B$ = 1000011101 y C = 0101011011, haced las operaciones que proponemos a continuación considerando que son números binarios en formato de signo y magnitud: $A + B$, $A - B$, $A + C$, $A - C$, $B - C$, $B + C$.

**25.** Repetid la actividad anterior considerando que las cadenas representan números en complemento a 2.

**26.** Si tenemos la cadena de bits 10110101, haced las conversiones siguientes:
**a)** Considerando que representa un número en Ca2, representad el mismo número en signo y magnitud y 16 bits.
**b)** Considerando que representa un número en signo y magnitud, representad el mismo número en Ca2 y 16 bits.

## 2.4. Números fraccionarios

Los números fraccionarios son los que tienen una parte más pequeña que la unidad, como por ejemplo el $0{,}03_{(10}$ o el $15{,}27_{(10}$. La representación de los números fraccionarios dentro de los computadores se suele hacer destinando un número fijo de bits, del total de bits del formato, a la representación de la parte

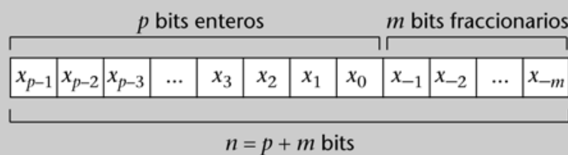
**Número decimal**

Usamos la expresión *número decimal* para designar un número en base 10, no un número con parte fraccionaria.

fraccionaria. Este tipo de representación recibe el nombre genérico de **representación de coma fija**.

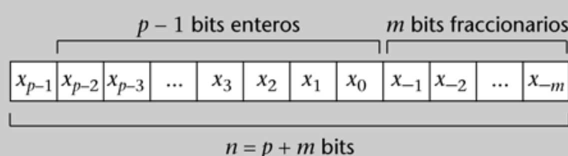
### Representación binaria de coma fija

Las representaciones de coma fija no almacenan la posición de la coma de manera explícita. Es en la definición del formato donde se especifica la posición de la coma, y se asume que siempre es la misma.

> En un formato de representación de coma fija de $n$ bits ($n = p + m$), donde $m$ bits son fraccionarios, los números representados son de la forma:
>
> 
> 
>
> La magnitud decimal del número fraccionario representado es $X_{(10} = \sum_{i=p-1}^{-m} x_i \cdot 2^i$, donde $x_i$ es el bit de la posición $i$-ésima.

### Signo y magnitud en coma fija

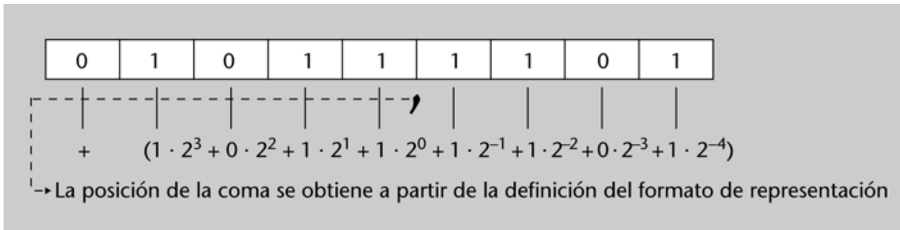
Las magnitudes fraccionarias también pueden llevar asociado un signo. En coma fija, lo más habitual es trabajar con una representación de signo y magnitud.

> En un formato de coma fija de $n$ bits ($n = p + m$), donde $m$ bits son fraccionarios, y en signo y magnitud, los números son de la forma:
>
> 
> 
>
> donde $x_{p-1}$ es el bit de signo y el resto de los bits codifican la magnitud.

**Otras maneras de representar números**

Hay otras maneras de representar números fraccionarios con signo. Por ejemplo, se podría emplear la representación de complemento a la base (Ca2 en binario), pero no es habitual.

Para conocer en decimal el valor numérico representado por la codificación 010111101 que está en signo y magnitud en un formato de 9 bits ($n$ = 9), de los cuales 4 son fraccionarios ($m$ = 4), tenemos que aplicar el TFN:




Por lo tanto, el número codificado es $+1011{,}1101_{(2} = 11{,}8125_{(10}$.

**Formato en coma fija**

Esta forma de representar los números fraccionarios es una extensión directa de la representación en signo y magnitud de números enteros. Por lo tanto, presenta las mismas ventajas e inconvenientes. Así, por ejemplo, el cero tiene dos representaciones, una con signo negativo y otra con signo positivo.

**Magnitud decimal de un número codificado en coma fija y signo y magnitud**

Para encontrar la magnitud decimal que representa la codificación 10010010 que está en un formato de 8 bits, 4 de los cuales son fraccionarios y en signo y magnitud, haremos las operaciones siguientes:

**1)** Separamos el bit de signo que es 1 (bit del extremo izquierdo) y que indica signo negativo. El resto de bits 0010010 codifica la magnitud.

**2)** El valor decimal de la magnitud lo podemos conocer aplicando el TFN:

$$0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = 2 + 0{,}125 = 1{,}125_{(10}$$

Por lo tanto, el número representado en decimal es $-1{,}125_{(10}$.

**Codificación de un valor decimal en coma fija y signo y magnitud**

Para codificar el número $-14{,}75_{(10}$, en un formato de coma fija de 8 bits donde 3 son fraccionarios y signo y magnitud haremos las operaciones siguientes:

**1)** Cambiar a base 2 el número $14{,}75_{(10}$, siguiendo el método basado en el teorema de la división entera.

Ved el método basado en el teorema de la división entera en el subapartado 1.3.2 de este módulo.

**a)** Codificar en binario la parte entera ($14_{(10}$) en 4 bits (puesto que de los 8 bits, 3 son fraccionarios y 1 codifica el signo; restan 4 por la parte entera), aplicando el algoritmo de divisiones sucesivas:

$$\begin{aligned} 14 &= 7 \cdot 2 + 0 \\ 7 &= 3 \cdot 2 + 1 \\ 3 &= 1 \cdot 2 + 1 \\ 1 &= 0 \cdot 2 + 1 \end{aligned} \quad \uparrow$$

Con el que obtenemos que $14_{(10} = 1110_{(2}$.

**b)** Codificar en binario la parte fraccionaria en 3 bits:

$$\begin{aligned} 0{,}75 \cdot 2 &= 1{,}50 = 1 + 0{,}5 \\ 0{,}50 \cdot 2 &= 1{,}0 = 1 + 0{,}0 \end{aligned} \quad \downarrow$$

Con el que obtenemos que $0{,}75_{(10} = 0{,}110_{(2}$

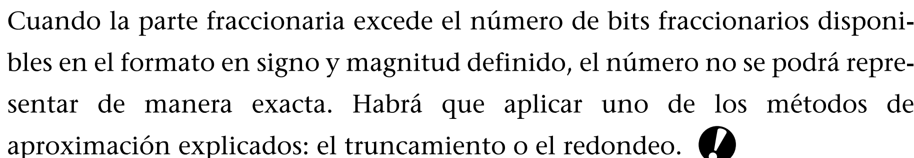
**c)** Juntar las partes entera y fraccionaria en el formato de 7 bits, 3 de los cuales son fraccionarios:

$$14{,}75_{(10} = 1110{,}110_{(2}$$

**2)** Añadir el bit de signo a la magnitud. El bit de signo es 1 puesto que el signo es negativo. La representación en un formato de coma fija de 8 bits donde 3 son fraccionarios y signo y magnitud del $-14{,}75_{(10}$ es el siguiente:

$$-14{,}75_{(10} = 11110{,}110_{(SM2}$$

Recordamos que la coma no se almacena, sino que una vez especificado el formato se conoce su posición. En realidad, un computador almacenaría el código 11110110 sin coma ni especificación de base, que están fijados en la definición del formato.



Cuando la parte fraccionaria excede el número de bits fraccionarios disponibles en el formato en signo y magnitud definido, el número no se podrá representar de manera exacta. Habrá que aplicar uno de los métodos de aproximación explicados: el truncamiento o el redondeo.

Ved las aproximaciones por truncamiento y redondeo en el subapartado 2.1.4 de este módulo.

A modo de ejemplo, intentemos representar el número $+8{,}9453125_{(10}$ en un formato de coma fija y signo y magnitud, con 8 bits de los cuales 3 son fraccionarios. Si hacemos el cambio de base, encontramos que $8{,}9453125_{(10} = 1000{,}1111001_{(2}$. Tendremos que usar uno de los métodos de aproximación, puesto que la parte fraccionaria no cabe en los 3 bits disponibles en el formato:

- Por **truncamiento**: Se trata, sencillamente, de despreciar los bits que no tienen cabida. El $1000{,}1111001_{(2}$ se aproximará por el $1000{,}111_{(2}$. Añadimos el bit de signo y la codificación final será 01000111. El error de representación que se comete en este caso es:

$$|1000{,}1111001_{(2} - 1000{,}111_{(2}| = 0{,}0001001_{(2} = 0{,}0703125_{(10}$$

- Por **redondeo**: Se suma la mitad de la precisión:

$$1000{,}1111001_{(2} + 0{,}0001_{(2} = 1001{,}0000001_{(2}$$

A continuación truncamos a 3 bits fraccionarios, de forma que el $1000{,}1111001_{(2}$ se aproximará por el $1001{,}000_{(2}$. Añadimos el bit de signo y la codificación final será 01001000. En este caso, el error de representación que se comete es menor:

$$|1000{,}1111001_{(2} - 1001{,}000_{(2}| = 0{,}0000111_{(2} = 0{,}0546875_{(10}$$

**Ejemplo**

Cambio de base 10 a base 2 del número $8{,}9453125_{(10}$:

1. Parte entera:

$8 = 4 \cdot 2 + 0$
$4 = 2 \cdot 2 + 0$
$2 = 1 \cdot 2 + 0$
$1 = 0 \cdot 2 + 1$

2. Parte fraccionaria:

$0{,}9453125 \cdot 2 = 0{,}890625 + 1$
$0{,}890625 \cdot 2 = 0{,}781250 + 1$
$0{,}781250 \cdot 2 = 0{,}5625 + 1$
$0{,}5625 \cdot 2 = 0{,}125 + 1$
$0{,}125 \cdot 2 = 0{,}25 + 0$
$0{,}25 \cdot 2 = 0{,}5 + 0$
$0{,}5 \cdot 2 = 0 + 1$

$8{,}9453125_{(10} = 1000{,}1111001_{(2}$

**La precisión**

La precisión es la distancia entre dos números representables consecutivos (ved el subapartado 2.1.2). Fácilmente, se puede comprobar que la precisión con 3 bits fraccionarios es $0{,}001_{(2}$ (distancia entre el 0,000 y el 0,001).

La mitad de este valor lo conseguimos desplazando la coma una posición a la izquierda (que equivale a dividir por 2 en base 2). Por lo tanto, la mitad de la precisión es $0{,}0001_{(2}$.

### Rango y precisión en coma fija

La magnitud binaria más grande que podemos representar en un formato de coma fija es la que se obtiene poniendo todos los bits que representan la magnitud a 1. Si el bit de signo lo ponemos a cero, tendremos la representación de la mayor magnitud positiva que se puede representar. Si el bit de signo es 1, se tratará de la mayor magnitud negativa que se puede representar. Estos números, el mayor y el menor, delimitan el intervalo que contiene los números que se pueden representar, es decir, el rango.

El número mayor representable en signo y magnitud y un formato de coma fija de 9 bits con 3 fraccionarios, es el $011111{,}111_{(2} = +31{,}875_{(10}$; el menor es el $111111{,}111_{(2} = -31{,}875_{(10}$. Por lo tanto, el rango en decimal de este formato es:

$$[-31{,}875_{(10}, +31{,}875_{(10}]$$

En general, el número mayor que se puede representar con *n* bits de los cuales *m* son fraccionarios se puede calcular suponiendo que todos los bits valen 1 y aplicando el TFN:

$$1 \cdot 2^{n-m-2} + 1 \cdot 2^{n-m-3} + .... + 1 \cdot 2^{0} + 1 \cdot 2^{-1} + .... + 1 \cdot 2^{-m} = 2^{n-m-1} - 2^{-m}$$

donde hemos aplicado la propiedad siguiente:

$$\overset{k}{111\cdots11} = 2^{k-1} + 2^{k-2} + \cdots 2^{1} + 2^{0} = \frac{2^{k} - 2^{0}}{2 - 1} = \frac{2^{k} - 1}{1} = 2^{k} - 1.$$

El **rango** de una representación en signo y magnitud y un formato en coma fija de *n* bits, donde *m* son fraccionarios, es

$$\left[-2^{n-m-1} + 2^{-m}, +2^{n-m-1} - 2^{-m}\right].$$

Del mismo modo, el **rango** de una representación de números fraccionarios sin signo en un formato de coma fija de *n* bits, donde *m* son fraccionarios, es el siguiente:

$$\left[0, +2^{n-m} - 2^{-m}\right].$$

### Ampliación del número de bits de un formato de coma fija

La ampliación de un formato en coma fija ha tener en cuenta los posibles cambios en la posición de la coma. De hecho, tan sólo hay que conocer cuántos bits se destinan a la ampliación de la parte fraccionaria y cuántos a la ampliación de la parte entera. Una ampliación de *k* bits de la parte fraccionaria comporta añadir *k* ceros a la derecha de la magnitud. Una ampliación de *p* bits de la parte entera se consigue si añadimos *p* ceros a la izquierda de la magnitud. Si trabajamos en signo y magnitud, tendremos que separar el signo de la magnitud para hacer los cambios y después añadirlo de nuevo al extremo izquierdo.

**Ejemplo**

Para ampliar en 3 bits la parte fraccionaria y en 2 bits la parte entera del número $111{,}001_{(SM2}$ que está en coma fija y signo y magnitud, añadiremos 3 ceros a la derecha de la magnitud y 2 ceros a la izquierda de la magnitud. La nueva codificación en el caso de signo y magnitud es 1**00**11,001**000**$_{(SM2}$, donde se marcan en negro los dígitos añadidos.

Esta ampliación, en caso de que el número $111{,}001_{(2}$ fuera una magnitud sin signo, daría lugar a la codificación **00**111,001**000**$_{(2}$.

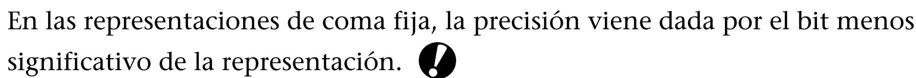
**La extensión** o ampliación de *k* bits por la parte fraccionaria y *p* bits por la parte entera de un formato de coma fija, tanto en signo y magnitud como sin signo, se consigue si añadimos *k* ceros a la derecha de la magnitud y *p* ceros a la izquierda de la magnitud.

### Precisión de un formato de coma fija

La precisión es la distancia más pequeña entre dos números representables consecutivos. Si trabajamos con una representación en coma fija de 3 bits donde 1 es fraccionario y signo y magnitud, los números que se pueden repre-

sentar son: $-1{,}1_{(2}$, $-1{,}0_{(2}$, $-0{,}1_{(2}$, $0{,}0_{(2}$, $+0{,}1_{(2}$, $+1{,}0_{(2}$ y $+1{,}1_{(2}$. Como podemos observar, todos ellos están separados por una distancia de $0{,}1_{(2}$. Por este motivo la precisión es $0{,}1_{(2}$.

En las representaciones de coma fija, la precisión viene dada por el bit menos significativo de la representación.



> La **precisión** de una representación en coma fija de *n* bits, donde *m* son fraccionarios, es $2^{-m}$.

**Suma y resta en coma fija**

Las operaciones de suma y de resta en coma fija se hacen a partir del algoritmo habitual descrito en el apartado 1.6, como podemos ver en los ejemplos siguientes:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | 0 | 0 | | | ← acarreo |
| | | 1 | , | 1 | 0 | 1 | (2 | |
| + | | 0 | , | 1 | 0 | 0 | (2 | |
| | 1 | 0 | , | 0 | 0 | 1 | (2 | ← resultado |

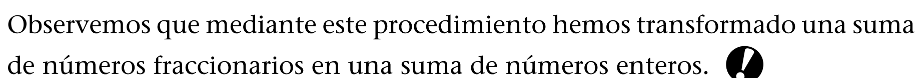
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | , | 1 | 0 | 1 | (2 | |
| | 0 | | 0 | 0 | | | ← acarreo |
| – | 0 | , | 1 | 0 | 0 | (2 | |
| | 1 | , | 0 | 0 | 1 | (2 | ← resultado |

Fijémonos en que un número fraccionario, como por ejemplo el $1{,}101_{(2}$, se puede escribir de la forma $1101_{(2} \cdot 2^{-3}$. Mediante este procedimiento podemos asociar un número entero, el $1101_{(2}$ en este caso, a un número fraccionario.

Si aplicamos esta transformación a los números $1{,}101_{(2}$ y $0{,}100_{(2}$ de la suma anterior, obtenemos: $1{,}101_{(2} = 1101_{(2} \cdot 2^{-3}$ y $0{,}100_{(2} = 0100_{(2} \cdot 2^{-3}$. La suma se puede hacer de la forma:

$$1101_{(2} \cdot 2^{-3} + 0100_{(2} \cdot 2^{-3} = (1101_{(2} + 0100_{(2)}) \cdot 2^{-3} = 10001_{(2} \cdot 2^{-3}$$

Observemos que mediante este procedimiento hemos transformado una suma de números fraccionarios en una suma de números enteros.



> Con la transformación anterior, las operaciones entre números fraccionarios se pueden llevar a cabo a través de operaciones entre números enteros.

Fijémonos también en el hecho de que en la representación no aparece la ubicación de la coma, pero, dado que todos los números con los que trabajaremos tendrán la coma en la misma posición, no hay que saber la posición para operar.

Es decir, las operaciones para sumar los números $11{,}01_{(2}$ y $00{,}01_{(2}$ o los números $1{,}101_{(2}$ y $0{,}001_{(2}$ son idénticas. La primera es:

$$11{,}01_{(2} + 00{,}01_{(2} = 1101_{(2} \cdot 2^{-2} + 0001_{(2} \cdot 2^{-2} = (1101_{(2} + 0001_{(2)}) \cdot 2^{-2}$$

y la segunda:

$$1{,}101_{(2} + 0{,}001_{(2} = 1101_{(2} \cdot 2^{-3} + 0001_{(2} \cdot 2^{-3} = (1101_{(2} + 0001_{(2)}) \cdot 2^{-3}$$

En ambos casos la operación realizada finalmente es la suma de los dos números enteros $1101_{(2}$ y $0001_{(2}$.

A continuación sumamos los números $11{,}111_{(2}$ y $01{,}111_{(2}$, que están en un formato de coma fija de 5 bits, donde 3 son fraccionarios:

```
  1 | 1 1 1 1        ← acarreo
    | 1 1 1 1 1 (2
+   | 0 1 1 1 1 (2
  ----------------
  1 | 0 1 1 1 0 (2   ← resultado
```

Para representar el resultado nos hace falta un dígito más de los que tenemos disponibles en el formato definido. Por eso, el resultado no es representable en el formato especificado. Se ha producido desbordamiento, que podemos reconocer de la misma forma que en la suma de números naturales.

**El desbordamiento en la suma de números sin signo en coma fija** se puede detectar de la misma forma que en el caso de la suma de números naturales. Recordamos que la resta no puede dar lugar a desbordamiento.

**El desbordamiento en la suma y la resta de números en coma fija y signo y magnitud** se puede detectar de la misma forma que en el caso de la suma y la resta de números enteros representados en signo y magnitud.

**La resta de números sin signo**

La resta de números sin signo no puede dar lugar a desbordamiento porque tenemos que restar la magnitud pequeña de la grande. Restar la magnitud grande de la magnitud pequeña no es una operación válida para números sin signo, porque el resultado tendría que ser un número con signo.

### Multiplicación y división por $2^k$ en coma fija binaria

Cuando la base de numeración es 2 en un sistema posicional de base fija, los pesos asociados a los dígitos son potencias de 2. Por consiguiente, multiplicar por 2 se traduce en aumentar en una unidad la potencia de 2 asociada a cada bit y dividir por 2 es equivalente a disminuir en una unidad la potencia de 2 asociada a cada bit.

Ved la multiplicación y la división por potencias de la base de numeración en el subapartado 1.8 de este módulo.

Aplicando el TFN, podemos escribir el número $0011{,}01_{(2}$ de la forma:

$$0011{,}01_{(2} = 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

Si multiplicamos por 2, se obtiene:

$$0011{,}01_{(2} \cdot 2 = (0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}) \cdot 2 =$$
$$= 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} = 00110{,}1_{(2}$$

En un formato de coma fija, la posición de la coma siempre es la misma y, por lo tanto, para multiplicar o dividir por la base desplazamos los bits a izquierda o derecha, añadiendo los ceros necesarios.

> **Multiplicar por $2^k$** un número en coma fija sin signo equivale a desplazar los bits *k* posiciones a la izquierda, completando los *n* bits del formato con la adición a la derecha de *k* ceros.
>
> **Dividir por $2^k$** un número en coma fija sin signo equivale a desplazar los bits *k* posiciones a la derecha, completando los *n* bits del formato con la adición a la izquierda de *k* ceros.

En coma fija y 6 bits, donde 2 son fraccionarios, el resultado de multiplicar $0011{,}01_{(2}$ por $2^2$ se obtiene desplazando los bits 2 posiciones a la izquierda y añadiendo 2 ceros a la derecha:

$$0011{,}01_{(2} \cdot 2 = 1101{,}\mathbf{00}_{(2}$$

Si el resultado no cabe en el formato, se produce desbordamiento. Esto se da cuando se pierden bits significativos (en coma fija sin signo, bits a 1).

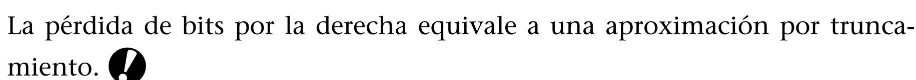
> Se produce **desbordamiento** al multiplicar un número sin signo en coma fija por $2^k$ cuando se pierden uno o más bits significativos (bits a 1) al desplazar los bits *k* posiciones.

La división por $2^k$ no produce desbordamiento, pero el cociente puede necesitar más bits fraccionarios que los disponibles en el formato:

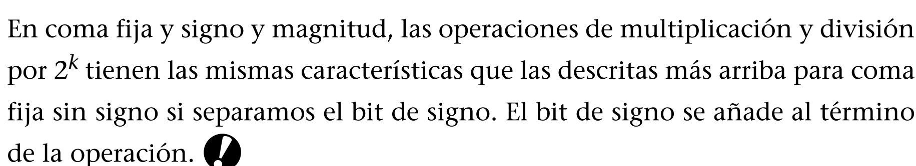
**El símbolo ≈**

El símbolo ≈ indica que se trata de una aproximación, no de una igualdad.

$$0101{,}00_{(2} / 2^4 \approx 0000{,}01_{(2} \qquad \text{(en decimal, } + 5_{(10} / 2^4 \approx + 0{,}25_{(10}\text{)}$$



La pérdida de bits por la derecha equivale a una aproximación por truncamiento.



En coma fija y signo y magnitud, las operaciones de multiplicación y división por $2^k$ tienen las mismas características que las descritas más arriba para coma fija sin signo si separamos el bit de signo. El bit de signo se añade al término de la operación.

## Actividades

**27.** Determinad qué valor decimal codifica la cadena de bits 1010010 en los supuestos siguientes:
**a)** Si se trata de un número en coma fija sin signo de 7 bits donde 4 son fraccionarios.
**b)** Si se trata de un número en coma fija sin signo de 7 bits donde 1 es fraccionario.

**28.** Codificad los números $+12{,}85_{(10}$, $+0{,}7578125_{(10}$ y $11{,}025_{(10}$ en una representación fraccionaria binaria en signo y magnitud de 8 bits donde 3 son fraccionarios. Utilizad una aproximación por redondeo en caso de que sea necesario.

**29.** Si tenemos una representación en coma fija binaria en signo y magnitud de 8 bits donde 3 bits son fraccionarios, determinad los números codificados por las cadenas de bits 01001111, 11001111, 01010100, 00000000 y 10000000.

**30.** Si las cadenas de bits 00101010, 11010010 y 10100010 representan números en coma fija sin signo de 8 bits donde 3 son fraccionarios, representadlos en un formato de coma fija sin signo de 12 bits donde 4 son fraccionarios.

**31.** Repetid la actividad anterior considerando que se trata de números en signo y magnitud.

**32.** Determinad el rango de representación y la precisión en los formatos siguientes:
**a)** Coma fija en signo y magnitud con 8 bits donde 3 son fraccionarios.
**b)** Coma fija en signo y magnitud con 8 bits donde 4 son fraccionarios.
**c)** Coma fija sin signo con 8 bits donde 3 son fraccionarios.
**d)** Coma fija sin signo con 8 bits donde 4 son fraccionarios.

**33.** Determinad la precisión necesaria para poder representar el número $+0{,}1875_{(10}$ de forma exacta (sin error de representación) con un formato de coma fija en base 2.

**34.** Determinad las características de rango y precisión, así como el número de dígitos enteros y fraccionarios necesarios en un formato de coma fija en signo y magnitud, para poder representar de forma exacta los números $+31{,}875_{(10}$ y $16{,}21875_{(10}$

**35.** Calculad la suma y la resta de los pares de números siguientes, asumiendo que están en coma fija en signo y magnitud con 8 bits donde 3 son fraccionarios. Verificad si el resultado es correcto:
**a)** $00111000_{(2}$ y $10100000_{(2}$
**b)** $10111010_{(2}$ y $11101100_{(2}$

# 3. Otros tipos de representaciones

El funcionamiento de los computadores actuales se basa en la electrónica digital, cuya característica distintiva es que toda la información con la que trabaja se codifica en base a dos únicos valores, que representamos simbólicamente con el 1 y el 0. Por lo tanto, todos los datos que procesa un computador digital tienen que estar representados exclusivamente por cadenas de unos y de ceros, es decir, por cadenas de bits. Entonces, el procesamiento de los datos consiste en aplicar operaciones aritméticas o lógicas a cadenas de bits.

En la sección precedente hemos expuesto las limitaciones inherentes a la tecnología empleada en los computadores digitales actuales y las formas más usuales en las que se codifican los valores numéricos. No ha sido una descripción exhaustiva de las formas de codificación de la información numérica, pero sí una muestra representativa de la manera como la información numérica se codifica para ser procesada dentro de los computadores digitales.

En los apartados siguientes se muestra, por un lado, cómo codificar información que inicialmente no es numérica, usando en último término los símbolos 1 y 0; y, por otro, algunos sistemas de numeración alternativos que tienen especial interés en determinadas circunstancias.

## 3.1. Representación de información alfanumérica

Se denomina *información alfanumérica* a la información no numérica constituida, básicamente, por el conjunto de letras, cifras y símbolos que se utilizan en las descripciones textuales y que reciben el nombre genérico de *caracteres*.

El número de caracteres empleado en los textos es relativamente grande: letras mayúsculas, letras minúsculas, vocales acentuadas, símbolos de puntuación, símbolos matemáticos, etc.

La representación de los caracteres se lleva a cabo asignando una cadena de bits única y específica para cada carácter, es decir, asignando a cada carácter un código binario. La asignación de códigos podría ser arbitraria, pero en la práctica es conveniente seguir unos criterios que faciliten el procesamiento de la información codificada. Por ejemplo, una asignación de códigos ascendente a las letras del alfabeto facilita la ordenación alfabética: el resultado de una sencilla operación de resta entre los códigos permitirá establecer el orden alfabético.

Siguiendo criterios que faciliten el tratamiento de los datos y con la intención de compatibilizar la información procesada por sistemas diferentes, se han es-

tandarizado unas pocas codificaciones, de entre las cuales, la codificación ASCII es la más extendida.

La codificación ASCII tiene una versión básica de 128 símbolos que forma el estándar *de facto* que usan la mayoría de los computadores, y una versión extendida, no tan estandarizada, que incluye 128 símbolos más. En la versión extendida se usan 8 bits para codificar los 256 símbolos que incluye, mientras que en la básica se usan 7.

En la tabla siguiente se puede ver la asignación de códigos ASCII*. La representación numérica asociada a cada símbolo se obtiene a partir de sus coordenadas. El índice de columna es el dígito decimal menos significativo, y el de fila, el más significativo. Por ejemplo, el carácter alfanumérico “3”, se representa por el valor decimal $51_{(10}$ (5u, d1), que en binario es el $00110011_{(2}$.

*** ASCII son las siglas de la expresión inglesa *American standard code for information interchange.***

| d \ u | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0u** | NULL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT |
| **1u** | LF | VT | FF | CR | SO | SI | DLE | DC1 | DC2 | DC3 |
| **2u** | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS |
| **3u** | RS | US | SP | ! | " | # | $ | % | & | ' |
| **4u** | ( | ) | * | + | , | - | . | / | 0 | 1 |
| **5u** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| **6u** | < | = | > | ? | @ | A | B | C | D | E |
| **7u** | F | G | H | I | J | K | L | M | N | O |
| **8u** | P | Q | R | S | T | U | V | W | X | Y |
| **9u** | Z | [ | \ | ] | ^ | _ | ` | a | b | c |
| **10u** | d | e | f | g | h | i | j | k | l | m |
| **11u** | n | o | p | q | r | s | t | u | v | w |
| **12u** | x | y | z | { | \| | } | ~ | DEL | | |

**Nota**

Es habitual usar las comillas para distinguir los números (3, 25, 234) de los caracteres o cadenas de caracteres (“3”, “25”, “234”).

Los primeros 31 códigos y el último no corresponden a símbolos del lenguaje o caracteres visibles (el carácter 32 –identificado como SP– representa el espacio en blanco). Estos códigos son caracteres de control utilizados para dar formato al texto o como comandos para los dispositivos periféricos (terminales alfanuméricos o gráficos, impresoras, etc.).

**Caracteres de control no visualitzables**

Algunos caracteres de control no visualizables son: DEL (borrar), ESC (escapada), HT (tabulador horizontal), LF (final de línea), CR (regreso a primera columna), FF (final de página), STX (inicio de texto) o ETX (final de texto).

Cada vez es más habitual que en los computadores se codifique texto en más de una lengua. Por este motivo, se ha ido popularizando la extensión de la codificación ASCII de los caracteres alfanuméricos a 2 bytes (16 bits), que utiliza el estándar llamado *Unicode* y que incluye los caracteres de las grafías más importantes.

Los códigos ASCII de 8 bits de los caracteres visibles (no así los correspondientes a caracteres de control) se pueden convertir a Unicode añadiendo 8 ceros a la izquierda para completar los 16 bits del estándar Unicode.

**El estándar Unicode**

El Unicode está estandarizado por la ISO/IEC (*International Organization for Standardization / International Electrotechnical Commission*) con el identificador 10646.

**Formato Unicode**

La codificación de textos en formato Unicode está presente en muchos de los procesadores de textos actuales, como por ejemplo, el Wordpad de Windows.

**Ejemplo de procesamiento de códigos ASCII**

Analizamos los códigos ASCII para averiguar cómo transformar el código de una letra mayúscula en su equivalente en minúscula. Los códigos consecutivos a partir del código 65 siguen el orden de las letras del alfabeto inglés tanto para las mayúsculas como, a partir del código 97, para las minúsculas. Por lo tanto, la distancia entre símbolos de mayúsculas y minúsculas es constante.

En concreto, el carácter "A" tiene el código 65, mientras que el carácter "a" tiene el 97. La diferencia entre los códigos es $32_{(10}$. Por lo tanto, para transformar el código ASCII de una letra mayúscula al código ASCII de la misma letra en minúscula, tenemos que sumar $32_{(10}$, al código ASCII en binario.

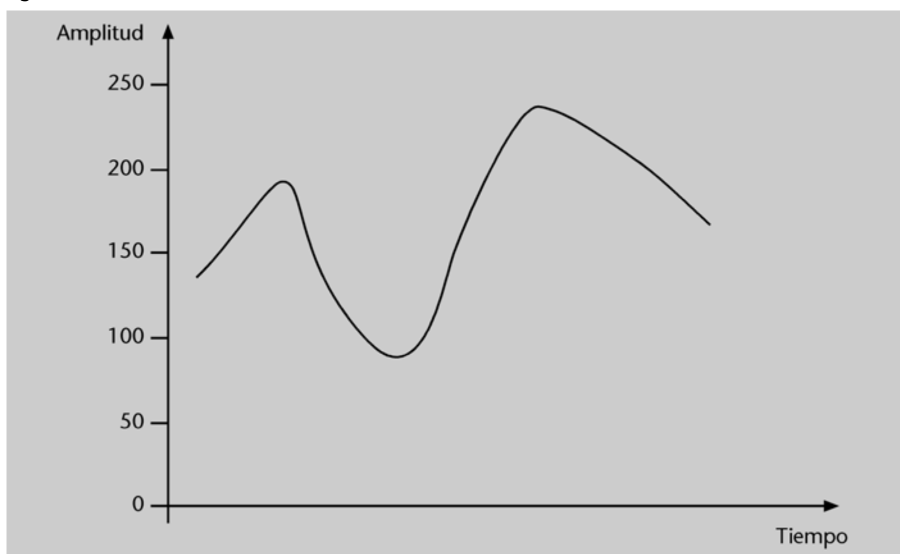
### 3.2. Codificación de señales analógicas

A veces, los datos que tiene que procesar un computador provienen de dispositivos que recogen información del entorno. Un micrófono, por ejemplo, capta las ondas sonoras que llegan hasta él. Los sensores de estos dispositivos son analógicos, es decir, generan una señal eléctrica de salida que se ajusta de manera continua a la variación del estímulo que reciben. El resultado es una señal eléctrica, cuya variación en el tiempo refleja la variación del estímulo que ha ido llegando al sensor del dispositivo.

Una señal eléctrica analógica es la que codifica la información mediante una variación continua de un parámetro eléctrico (tensión, frecuencia, intensidad) que se ajusta de manera proporcional al estímulo original.

La figura 1 muestra una señal analógica, en la que la amplitud varía de forma continua en el tiempo:

Figura 1



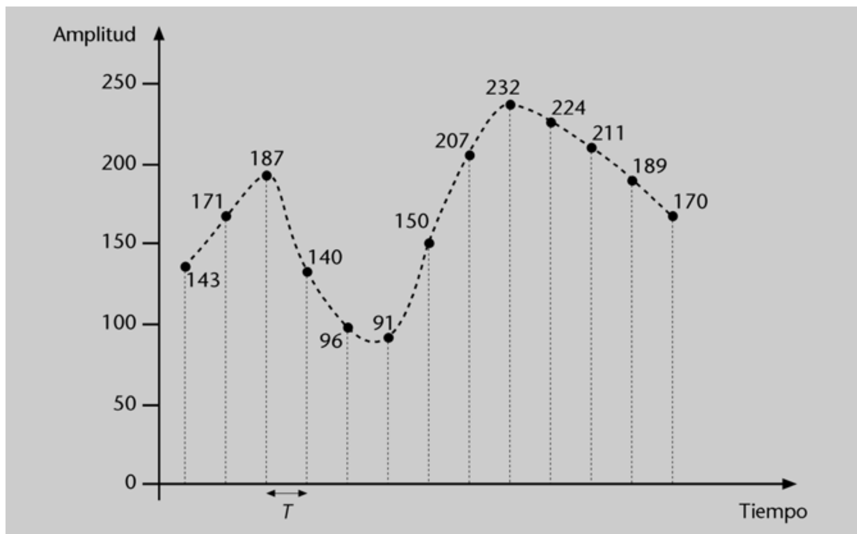
Los procesadores digitales no pueden tratar directamente las señales analógicas y, como en el caso de los números o de los caracteres alfanuméricos, se tienen que codificar utilizando únicamente los símbolos 1 y 0. El proceso que permite esta transformación es la **digitalización**.

La **digitalización** consiste en convertir una representación analógica a una representación digital binaria, es decir, a una secuencia ordenada de números binarios.

El proceso de digitalización consta de tres etapas, que son el muestreo, la cuantificación y la codificación binaria:

**1)** El **muestreo** (o **discretització**) consiste en tomar muestras de la señal analógica a intervalos de tiempos regulares.

Figura 2

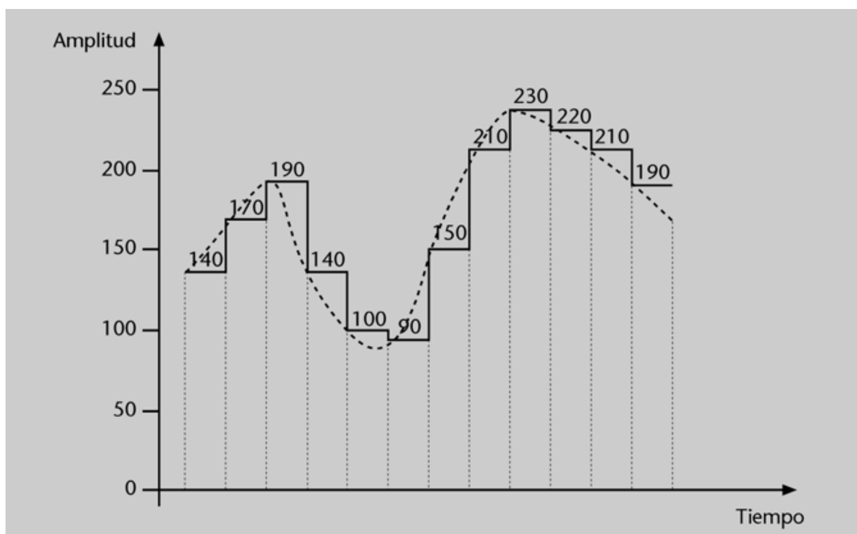



**Muestreo de una señal analógica**

Un muestreo de la señal dibujada en la figura 1 a intervalos de período *T* daría el resultado que se ve a la figura 2.

**2)** La **cuantificación** consiste en asignar un valor, de entre un conjunto finito, a la amplitud de la señal en cada intervalo de muestreo.

Figura 3




**Cuantificación de una señal**

Si admitimos únicamente valores múltiples de 10, la cuantificación del muestreo de la figura 2 da lugar a la secuencia de valores numéricos: 140, 170, 190, 140, 100, etc. que representa una aproximación "escalonada" a la señal continua original, como se muestra en la figura 3.

Cuanto más pequeños sean los intervalos de tiempo del muestreo (hasta un cierto límite más allá del cual ya no ganamos nada), y cuanto mayor sea el conjunto de valores admitidos en la cuantificación, más cercana será la información digitalizada a la información analógica original.

**3)** La **codificación binaria** consiste en traducir los valores de las muestras a un sistema binario, es decir, expresar los valores mediante ceros y unos.

**Ventajas de la digitalización**

La digitalización es la técnica que se usa, por ejemplo, para grabar música en un CD. El sonido se muestrea, se codifica en binario y se graba en el CD haciendo muescas: un 1 se traduce en hacer una muesca, un 0 se traduce en no hacer muesca.

La tecnología actual permite hacer el muestreo y la cuantificación suficientemente acotados para que la distancia entre los escalones provocados por la digitalización no sean perceptibles por el oído humano. Por otro lado, la digitalización evita los ruidos y distorsiones que se introducen con medios analógicos, cosa que permite que el sonido digital sea de más calidad que el analógico.

**Ejemplo de codificación binaria**

Los valores de las amplitudes que aparecen en la figura 3 van desde el 90 hasta el 230. Podemos codificar este rango de valores decimales en binario usando 8 bits (puesto que $2^7 < 230 < 2^8$). La tabla siguiente muestra la codificación en binario de los valores decimales de la cuantificación de la figura 3:

| Amplitud | Codificación binaria |
|---|---|
| 90 | 01011010 |
| 100 | 01100100 |
| 140 | 10001100 |
| 150 | 10010110 |
| 170 | 10101010 |
| 190 | 10111110 |
| 210 | 11010010 |
| 220 | 11011100 |
| 230 | 11100110 |

De hecho, conviene tener en cuenta que en la cuantificación sólo se han permitido múltiplos de 10. Así, en los valores de amplitud podemos despreciar el cero de la derecha y considerar el rango de valores [9, 23]. Para codificar en binario los números dentro de este rango son suficientes 5 bits (puesto que $2^4 < 23 < 2^5$):

| Amplitud | Codificación binaria |
|---|---|
| 90 | 01001 |
| 100 | 01010 |
| 140 | 01110 |
| 150 | 01111 |
| 170 | 10001 |
| 190 | 10011 |
| 210 | 10101 |
| 220 | 10110 |
| 230 | 10111 |

Esta codificación es más eficiente, porque utiliza un menor número de bits. Usando esta codificación, la información que en la figura 3 se expresaba mediante una línea curva ahora se expresa por la secuencia de códigos binarios siguiente:

01110
10001
10011

01110
01010
01001
01111
10101
10111
10110
10101
10011

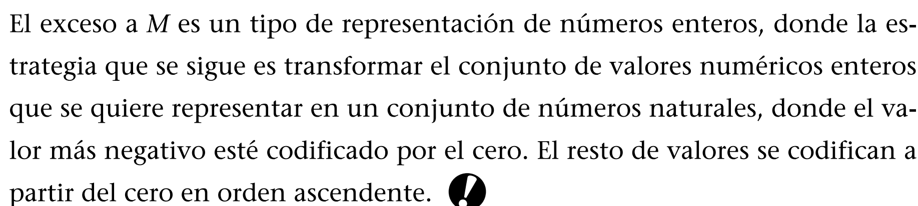
Esta secuencia de valores binarios constituye una aproximación escalonada a la curva continua de la figura 1.

La digitalización permite transformar en números cualquier señal analógica de nuestro entorno y conseguir, así, que se pueda procesar dentro de un computador digital.

### 3.3. Otras representaciones numéricas

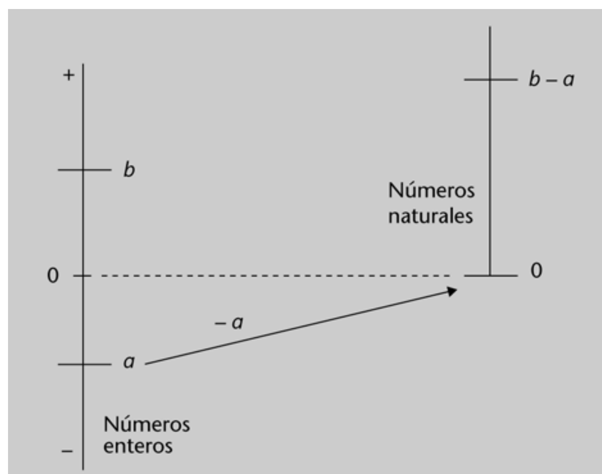
La sección 2 está dedicada a la descripción de las codificaciones más usuales de números enteros y fraccionarios tanto con signo como sin signo. Allí se han descrito las codificaciones más utilizadas para representar información numérica dentro de los computadores, sin embargo, hay algunas representaciones más que conviene conocer y que se describen en los apartados siguientes.

#### 3.3.1. Representación en exceso a *M*



El exceso a $M$ es un tipo de representación de números enteros, donde la estrategia que se sigue es transformar el conjunto de valores numéricos enteros que se quiere representar en un conjunto de números naturales, donde el valor más negativo esté codificado por el cero. El resto de valores se codifican a partir del cero en orden ascendente.

La figura siguiente muestra gráficamente esta estrategia:



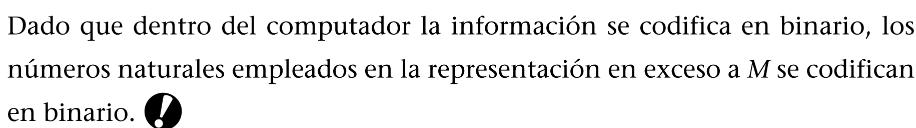
Consideremos el intervalo [–5, +5]. Para desplazar este rango de valores enteros a un conjunto de valores naturales, sólo tenemos que sumar 5 a cada número entero del intervalo. De esta forma, los números pasan a estar en el intervalo [0, 10]. Con este desplazamiento, el valor entero –5 da lugar al valor natural 0, puesto que –5 – (–5) = –5 +5 = 0; el valor +2 da lugar al valor natural 7, puesto que +2 – (–5) = +2 +5 = 7; etc.

> El intervalo [*a* , *b*] de números enteros se puede desplazar al intervalo de números naturales [0, *b* – *a* ] restando *a* a cada entero del intervalo [*a* , *b*].

Este tipo de estrategia se denomina representación en **exceso a *M***, donde *M* es el desplazamiento que se aplica al intervalo de enteros que se quiere codificar.

> La **representación en exceso a *M*** de un número entero *X* se obtiene sumando el desplazamiento *M* al valor numérico *X*. Por consiguiente, encontraremos el valor de un número codificado en exceso a *M*, restando *M* a la codificación.

A modo de ejemplo, podemos decir que en una representación en exceso a 10, el número entero –4 se representa mediante el número natural 6 (puesto que –4 + 10 = 6), y que el 0 se representa mediante el número 10 (puesto que 0 + 10 = 10).



Dado que dentro del computador la información se codifica en binario, los números naturales empleados en la representación en exceso a *M* se codifican en binario.

**Representación en exceso a *M***

Para representar el valor $-6_{(10}$ en exceso a 7 y 4 bits procederemos de la forma siguiente:

**1)** Sumamos el desplazamiento para encontrar la codificación en decimal $-6_{(10} + 7_{(10} = 1_{(10}$.
**2)** Codificamos en binario y 4 bits el número obtenido $1_{(10} = 0001_{(2}$.

El valor $-6_{(10}$ se codifica en exceso a 7 y 4 bits como 0001.

Para saber qué valor representa la cadena de bits 1100 que está codificada en exceso a 7 y 4 bits,procederemos de la forma siguiente:

**1)** Hacemos un cambio de base para encontrar la codificación en decimal $1100_{(2} = 12_{(10}$.
**2)** Restamos el desplazamiento para encontrar el valor codificado $12_{(10} - 7_{(10} = +5_{(10}$.

El valor que representa la cadena de bits 1101 codificada en exceso a 7 y 4 bits es el $+5_{(10}$.

El exceso a *M* es un tipo de representación de números enteros empleado para codificar el valor del exponente cuando se trabaja en coma flotante.

### 3.3.2. Representación en coma flotante

A menudo se tienen que representar números muy grandes (como por ejemplo la velocidad de transmisión de la luz en el vacío $c$=299792500 m/s) o bien números muy pequeños (como la masa de un electrón $m_e$ = 0,000000000000000000000000000091095 kg), y quizás de forma simultánea. Para evitar el uso de un gran número de dígitos en la representación de estos números, se emplea el formato de **coma flotante**.

**Coma flotante**

La ventaja de la coma flotante es la capacidad de representar con pocos dígitos números que en otros formatos necesitan muchos dígitos para ser representados.

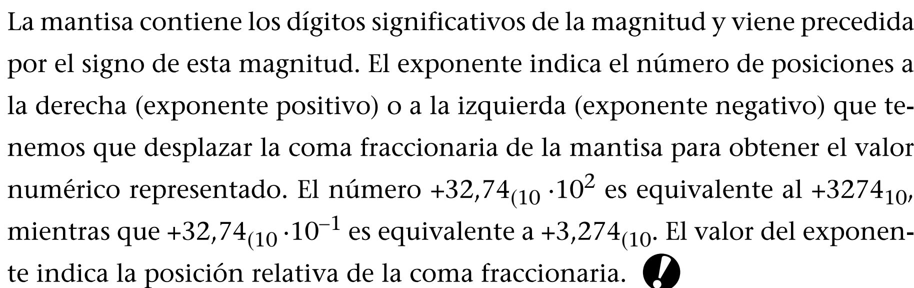
Los números en **coma flotante** toman la forma:

$$\pm R \cdot b^e$$

donde + o – indica el signo de la magnitud representada, $R$ es un número fraccionario que recibe el nombre de **mantisa** , $b$ es la base de numeración y $e$ es un número entero que recibe el nombre de **exponente**.

**Terminología**

La **coma flotante** también recibe el nombre de **notación científica**.



La mantisa contiene los dígitos significativos de la magnitud y viene precedida por el signo de esta magnitud. El exponente indica el número de posiciones a la derecha (exponente positivo) o a la izquierda (exponente negativo) que tenemos que desplazar la coma fraccionaria de la mantisa para obtener el valor numérico representado. El número $+32{,}74_{(10} \cdot 10^2$ es equivalente al $+3274_{10}$, mientras que $+32{,}74_{(10} \cdot 10^{-1}$ es equivalente a $+3{,}274_{(10}$. El valor del exponente indica la posición relativa de la coma fraccionaria.

**Atención**

A lo largo del texto usaremos las letras ***S***, ***e*** y ***R*** para referirnos, respectivamente, al signo, el exponente y la mantisa.

En el trabajo manual adaptamos dinámicamente el número de dígitos empleados en la mantisa y el exponente de la representación en coma flotante. Podemos utilizar $+2{,}995 \cdot 10^8$ o $+0{,}02995 \cdot 10^{10}$ según convenga a nuestras necesidades. Dentro de los computadores tenemos que ceder esta flexibilidad y adoptar restricciones que simplifiquen el procesamiento de datos. Por eso, se asume que la base de numeración es 2 y que el número de bits destinados a la mantisa y al exponente se fija en la especificación del formato.

**Símbolo +**

El símbolo + no suele aparecer delante del exponente o de la mantisa cuando son positivos.

Para la representación en coma flotante dentro de los computadores se asume que **la base de numeración es 2**, y que la definición del formato fija el número de bits de la mantisa y el número de bits del exponente.

Por otro lado, la representación de un valor numérico en coma flotante no es única. Por ejemplo, algunas representaciones en coma flotante del número $26300_{(10}$ son: $2{,}63_{(10} \cdot 10^4$, $0{,}263_{(10} \cdot 10^5$, $263_{(10} \cdot 10^2$, $2630_{(10} \cdot 10^1$, $26300_{(10} \cdot 10^0$ o $263000_{(10} \cdot 10^{-1}$. De nuevo, simplificamos el tratamiento de estos números, si se restringe esta flexibilidad, fijando el formato de la mantisa. Por ejemplo, el formato puede determinar que la coma está a la derecha del primer dígito no nulo. Con esta limitación, el $26300_{(10}$ tiene una representación única que es $2{,}63 \cdot 10^4$.

**Nota**

En coma flotante, la representación de un valor numérico no es única.

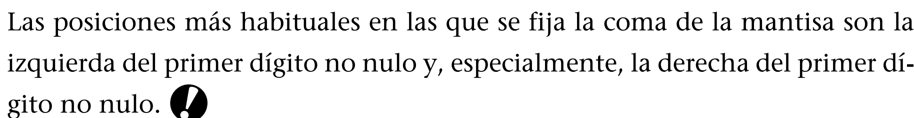
**Nota**

Fijando la posición de la coma de la mantisa se facilita la comparación de números.

Para evitar la multiplicidad de representaciones de un valor numérico, propia de la representación en coma flotante, en la definición del formato **se fija la posición de la coma fraccionaria respecto al primer dígito no nulo** de la mantisa.

Cuando la mantisa tiene fijada la posición de la coma fraccionaria, recibe el calificativo de **mantisa normalizada**.

Las posiciones más habituales en las que se fija la coma de la mantisa son la izquierda del primer dígito no nulo y, especialmente, la derecha del primer dígito no nulo.



Los valores numéricos representados en coma flotante con mantisa normalizada y coma a la derecha del primer bit no nulo son de la forma:

$$\pm 1, x_{-1} x_{-2} \cdots x_{-k} \cdot 2^e$$

donde $x_i$ son dígitos binarios (bits), 2 es la base de numeración en decimal y $e$ es el exponente.

**Coma flotante con mantisa normalizada**

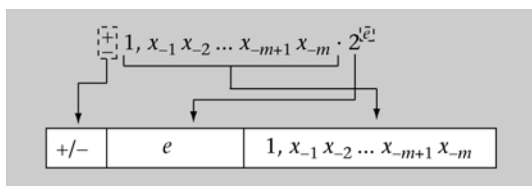
En las representaciones en coma flotante con la coma de la mantisa fijada a la izquierda del primer dígito no nulo, los números son de la forma:

$$\pm 0,1 x_{-2} \cdots x_{-k} \cdot 2^e \cdot$$

La codificación en coma flotante tiene que incorporar la información de signo (habitualmente 0 para los positivos y 1 para los negativos), el valor de la mantisa y el del exponente. En cambio, no se guarda la base de numeración dado que se asume que siempre es 2. La figura siguiente muestra el orden en el que usualmente se disponen estos valores:

**El bit de signo**

Como norma, el bit de signo es 0 para números positivos y 1 para números negativos.




**Signo-exponente-mantisa**

El orden de precedencia signo-exponente-mantisa no es el único posible, pero sí el más ampliamente utilizado.

En coma flotante, el exponente suele estar restringido a los enteros. Para codificarlo, lo más habitual es usar exceso a $M$, donde $M$ toma frecuentemente los valores $2^{q-1}$ o $2^{q-1} - 1$, donde $q$ es el número de bits del exponente.

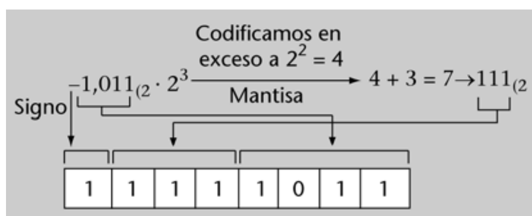
**Representación del exponente**

Por el exponente, el más extendido es el uso de exceso a $M$, pero se podría usar otro tipo de codificación, como por ejemplo Ca2.

En coma flotante de 8 bits, mantisa normalizada de 4 bits y exponente en exceso a $M$, donde $M$ toma el valor $2^{q-1}$ y $q$ es el número de bits disponibles para la representación del exponente, la codificación del número $-10,11_{(2} \cdot 2^2$ es la que se muestra a la figura siguiente:

**Nota**

De los 8 bits, 4 son por la mantisa y 1 por el signo. Restan 3 para el exponente. Si $M$ es $2^{q-1}$, $M = 2^{3-1} = 2^2 = 4$.
El exponente se codifica en exceso a 4.



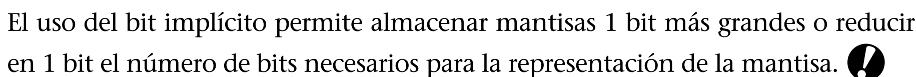
La tabla siguiente muestra algunos números representados en este formato:

| Números | S | e | | | R | | | |
|---|---|---|---|---|---|---|---|---|
| $-1{,}101_{(2}\cdot 2^{-1}$ | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| $1{,}101_{(2}\cdot 2^{-1}$ | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| $1{,}0_{(2}\cdot 2^{-3}$ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| $-1{,}10_{(2}\cdot 2^{+1}$ | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |

En la tabla precedente, podemos observar que el primer bit de la mantisa (columna *R*) siempre es 1, porque se codifican de la forma $1,x_{-1}x_{-2}\cdots$. Tienen una parte fija (1,) y una parte variable ( $x_{-1}x_{-2}\cdots$ ). Para optimizar recursos, podemos almacenar sólo la parte variable, puesto que la parte fija es conocida y común a todos los números. Esto permitirá almacenar 1 bit más para la mantisa, aumentando así su precisión, o bien utilizar un bit menos en la representación. Las mantisas almacenadas aplicando esta técnica reciben el nombre de **mantisas con bit implícito**.

La técnica del **bit implícito** consiste en almacenar sólo la parte variable de las mantisas normalizadas y asumir la parte fija como conocida y definida en el formato de la representación.

El uso del bit implícito permite almacenar mantisas 1 bit más grandes o reducir en 1 bit el número de bits necesarios para la representación de la mantisa.



**Codificación de un número decimal en coma flotante normalizada y binaria**

Para codificar el número $+104_{(10}$ en un formato de coma flotante normalizada de 8 bits, de los cuales 3 bits se destinan a la mantisa con bit implícito y exponente en exceso a *M*, seguiremos el proceso siguiente:

**1)** Codificar el número $+104_{(10}$ en base 2.

Si aplicamos el método de cambio de base basado en divisiones sucesivas, obtenemos que $104_{(10} = 1101000_{(2}$.

**2)** Normalizar la mantisa de la forma $1,x_{-1}x_{-2}x_{-3}\ldots$: $1101000_{(2} = 1{,}101 \cdot 2^6$

**3)** Identificar el signo, el exponente y la mantisa.

**a)** El número es positivo; por lo tanto, el bit de signo será 0: $S = 0$.
**b)** La mantisa de este número es 1,101. El formato indica que trabajamos con una mantisa de 3 bits y bit implícito. Por lo tanto, guardaremos los 3 bits a la derecha de la coma: 101.
**c)** El exponente toma el valor 6.

**4)** Codificar en exceso a *M* el exponente. De los 8 bits del formato, 3 se usan para la mantisa y 1 para el signo. Restan 4 para el exponente. Por lo tanto, el valor del exceso es $2^{4-1} = 2^3 = 8$. El $6_{(10}$ codificado en exceso a 8 es $6_{(10} + 8_{(10} = 14_{(10}$. Si aplicamos otra vez el método de cambio de base basado en divisiones sucesivas, encontramos que el $14_{(10}$ en base 2 es el $1110_{(2}$. Por lo tanto, $e = 1110$.

**5)** Unir las codificaciones de signo, exponente y mantisa en la orden de precedencia correcto ($S - e - R$) para obtener la representación final:

| S | Exponente | | | | Mantisa | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

**Cambio de base**

Para cambiar a base 2 el $104_{(10}$ hacemos divisiones sucesivas:

$104 = 52 \cdot 2 + 0$
$52 = 26 \cdot 2 + 0$
$26 = 13 \cdot 2 + 0$
$13 = 6 \cdot 2 + 1$
$6 = 3 \cdot 2 + 0$
$3 = 1 \cdot 2 + 1$
$1 = 0 \cdot 2 + 1$

$104_{(10} = 1101000_{(2}$

**Cambio de base**

Para cambiar a base 2 el $14_{(10}$ hacemos divisiones sucesivas:

$14 = 7 \cdot 2 + 0$
$7 = 3 \cdot 2 + 1$
$3 = 1 \cdot 2 + 1$
$1 = 0 \cdot 2 + 1$

$14_{(10} = 1110_{(2}$

Por lo tanto, la codificación del número $+104_{(10}$ en el formato binario de coma flotante especificado es 0<u>111</u>0101*.

* El subrayado destaca la posición del exponente dentro de una cadena de bits que codifica un número en coma flotante.

**Descodificación de un número en coma flotante normalizada binaria**

Para hallar el valor decimal del número 01010101, que está en un formato de coma flotante normalizada de 8 bits con 4 bits de mantisa con bit implícito y exponente en exceso, seguiremos el proceso siguiente:

**1)** Identificar el signo.
Si asumimos que el formato mantiene signo, exponente y mantisa en este orden, el bit del extremo izquierdo codifica el signo. En el 01010101 el primer bit es 0; por lo tanto, el signo es positivo.

**2)** Identificar la mantisa.
Dado que la mantisa ocupa las 4 posiciones más bajas, se trata de los bits 0101. Ahora bien, el formato indica la existencia de bit implícito. Por lo tanto, la mantisa es en realidad $1{,}0101_{(2}$.

**3)** Identificar el exponente.
El exponente está determinado por los 3 bits que quedan:

| S | Exponente | | | Mantisa | | | |
|---|---|---|---|---|---|---|---|
| 0 | **1** | **0** | **1** | 0 | 1 | 0 | 1 |

$M$ toma el valor $2^{q-1}$, por lo cual $M = 2^{3-1} = 2^2 = 4$. Así, el exponente codificado es $101_{(2} - 4_{(10} = 5_{(10} - 4_{(10} = 1_{(10}$. El exponente $e = 1_{(10}$.

**4)** Unir signo, exponente y mantisa.
Signo positivo, mantisa $1{,}0101_{(2}$ y exponente $1_{(10}$: el número representado es el $+1{,}0101 \cdot 2^1$.

**5)** Hacer un cambio de base con objeto de hallar el valor decimal.
Si aplicamos el TFN, tenemos que:

$$+1{,}0101_{(2}\cdot 2^1 = (1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4}) \cdot 2^1 = 2^1 + 2^{-2}\cdot 2^1 + 2^{-4}\cdot 2^1 = +2{,}625_{(10}$$

Por lo tanto, el número $0\underline{101}0101_{(2}$ codifica en el formato de coma flotante especificado el valor decimal $+2{,}625_{(10}$.

Para hallar el valor decimal del número $10010001_{(2}$, que está en un formato de coma flotante normalizada de 8 bits con 5 bits de mantisa con bit implícito y exponente en exceso, seguiremos el proceso siguiente:

**1)** Identificar el signo.
Si asumimos que el formato mantiene signo, exponente y mantisa en este orden, el bit del extremo izquierdo codifica el signo. En el 10010001 el primer bit es 1; por lo tanto, el signo es negativo.

**2)** Identificar la mantisa.
Dado que la mantisa ocupa las 5 posiciones más bajas, se trata de los bits 10001. El formato indica la existencia de bit implícito. Por lo tanto, la mantisa es, en realidad, $1{,}10001_{(2}$.

**3)** Identificar el exponente.
El exponente está determinado por los 2 bits que quedan:

| S | *e* | | Mantisa | | | | |
|---|---|---|---|---|---|---|---|
| 1 | **0** | **0** | 1 | 0 | 0 | 0 | 1 |

$M$ toma el valor $2^{q-1}$, por lo cual $M = 2^{2-1} = 2^1 = 2$. Así, el exponente codificado es $00_{(2} - 2_{(10} = 0_{(10} - 2_{(10} = -2_{(10}$. El exponente $e = -2_{(10}$.

**4)** Unir signo, exponente y mantisa.
Signo negativo, mantisa $1{,}10001_{(2}$ y exponente $-2_{(10}$: el número representado es el $-1{,}10001_{(2} \cdot 2^{-2}$.

**5)** Hacer un cambio de base para encontrar el valor decimal.

Si aplicamos el TFN, tenemos que:

$$-1{,}10001_{(2}\cdot 2^{-2} = -(1\cdot 2^{0} + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 0\cdot 2^{-4} + 1\cdot 2^{-5})\cdot 2^{-2} =$$
$$= -(2^{-2} + 2^{-1}\cdot 2^{-2} + 2^{-5}\cdot 2^{-2}) = -0{,}3828125_{(10}$$

Por lo tanto, el número $1\underline{00}10001_{(2}$ codifica en el formato de coma flotante especificado el valor decimal $-0{,}3828125_{(10}$.

### 3.3.3. Representación BCD

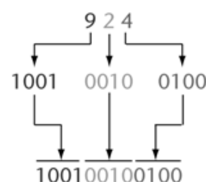
Una estrategia alternativa para la representación de números es la codificación directa de los dígitos decimales, sin hacer un cambio de base. Como en el caso de la codificación de la información alfanumérica, tenemos que asignar un código binario a cada dígito decimal. La tabla siguiente muestra la codificación binaria de los dígitos decimales en BCD:

| Dígito decimal | Codificación binaria |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

**BCD**

BCD son las siglas de *binary coded decimal,* es decir, decimal codificado en binario.

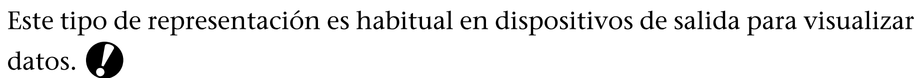
Los códigos BCD de los dígitos decimales son de 4 bits. La figura siguiente muestra la forma en que podemos representar un número decimal si codificamos cada dígito individualmente, lo que se denomina **representación BCD**:



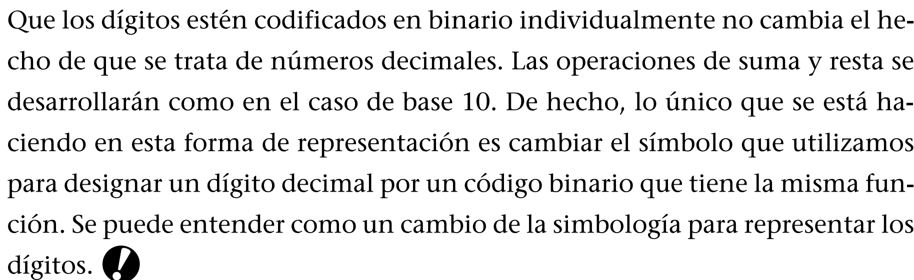

La **representación BCD** (*binary coded decimal*) consiste en codificar los números decimales dígito a dígito. Cada dígito decimal se sustituye por 4 bits que corresponden a la codificación en binario del dígito decimal.

La codificación binaria dígito a dígito de los números decimales aprovecha parcialmente la capacidad de representación. El código 1100, por ejemplo, no

se usa. De las 16 combinaciones posibles que se pueden hacer con 4 bits, sólo se usan 10. Por consiguiente, la representación de un número en BCD necesita más bits que en binario.



Este tipo de representación es habitual en dispositivos de salida para visualizar datos.



Que los dígitos estén codificados en binario individualmente no cambia el hecho de que se trata de números decimales. Las operaciones de suma y resta se desarrollarán como en el caso de base 10. De hecho, lo único que se está haciendo en esta forma de representación es cambiar el símbolo que utilizamos para designar un dígito decimal por un código binario que tiene la misma función. Se puede entender como un cambio de la simbología para representar los dígitos.

### Actividades

**36.** Codificad en BCD el número $125_{(10}$.

**37.** Codificad en BCD el número $637_{(10}$.

**38.** Indicad qué número codifica la representación BCD siguiente 00010011100.

**39.** Codificad el número $427_{(10}$ en BCD y en binario. Comparad el número de bits necesario en los dos casos.

**40.** Hallad el valor decimal que codifican las cadenas de bits siguientes, interpretando que se trata de números en un formato de coma flotante de 8 bits con mantisa normalizada de la forma 1,*X* y con bit implícito:
**a)** 11110010, donde la mantisa es de 4 bits.
**b)** 01010011, donde la mantisa es de 3 bits.

**41.** Haced las codificaciones siguientes:
**a)** El número $-1{,}335_{(10}$ en coma flotante de 8 bits, mantisa de 3 bits normalizada de la forma 1,*X* y con bit implícito empleando una aproximación por truncamiento.
**b)** Repetid el apartado anterior, pero con una aproximación por redondeo.
**c)** El número $10{,}0327_{(10}$ en coma flotante de 9 bits, mantisa de 3 bits normalizada de la forma 1,*X* y con bit implícito empleando una aproximación por truncamiento.

**42.** Determinad si el número $\mathbf{2{,}89_{(10} \cdot 10^{10}}$ es representable en un formato de coma flotante de 16 bits, con mantisa normalizada de la forma 1,*X*, bit implícito y 5 bits para el exponente.

**43.** Determinad si el número $\mathbf{-1256_{(10} \cdot 10^{-2}}$ es representable en un formato de coma flotante de 10 bits, con mantisa normalizada de la forma 1,*X*, bit implícito y 6 bits para el exponente.

# Resumen

En este módulo se presenta un análisis de los sistemas de numeración posicionales y se exponen las formas de representar valores numéricos que es habitual utilizar dentro de los computadores. Los puntos principales que se tratan en este módulo son:

- El TFN y el algoritmo de divisiones sucesivas que permiten cambiar entre bases diferentes la representación de un valor numérico.

- La representación de números naturales mediante representaciones posicionales empleando base 2 (binario), base 16 (hexadecimal) y base 10 (decimal), así como las operaciones de suma y resta de números naturales.

- Las limitaciones derivadas de los condicionamientos físicos de los computadores y las características que presentan los diferentes formatos de representación (rango y precisión), así como el fenómeno del desbordamiento y las técnicas de aproximación.

- La codificación de los números enteros empleando las representaciones en complemento a 2 y signo y magnitud, y las operaciones de suma y resta en cada una de estas codificaciones.

- La codificación de números fraccionarios con y sin signo en coma fija.

- El empaquetamiento de información en hexadecimal y la codificación en BCD.

# Ejercicios de autoevaluación

**1.** Codificad en complemento a 2 y signo y magnitud el número $-10_{(10}$ empleando 8 bits.

**2.** Determinad el valor decimal que codifica la cadena de bits 00100100 en los supuestos siguientes:
**a)** Si se trata de un número codificado en complemento a 2.
**b)** Si se trata de un número codificado en signo y magnitud.

**3.** Sumad en binario los números $111010101100_{(2}$ y $11100010010_{(2}$. Analizad el resultado obtenido.

**4.** Codificad en un formato de coma fija de 8 bits donde 3 son fraccionarios y en signo y magnitud, el número $+12{,}346_{(10}$. Emplead una aproximación por truncamiento en caso de que sea necesario.

**5.** Codificad en un formato de 8 bits y complemento a 2 el número $-45_{(10}$.

**6.** Determinad el número mínimo de bits enteros y fraccionarios necesarios en coma fija y signo y magnitud para codificar el número $-35{,}25_{(10}$.

**7.** Codificad los números $+12{,}25_{(10}$ y el $+32{,}5_{(10}$ en un formato de coma fija y signo y magnitud de 9 bits donde 2 son fraccionarios y sumadlos.

**8.** Codificad en la representación BCD el número $178_{(10}$.

**9.** Codificad en un formato de coma flotante de 8 bits con mantisa de 3 bits normalizada de la forma $1,X$ con bit implícito, y exponente en exceso a $M$ con $M = 2^{q-1}$ (donde $q$ es el número de bits del exponente), el número $+12{,}346_{(10}$. Emplead una aproximación por truncamiento en caso de que sea necesario.

**10.** Determinad el valor decimal que representa el código $378_{(16}$, sabiendo que se trata de un número en coma flotante de 12 bits con mantisa de 4 bits normalizada de la forma $1,X$ y bit implícito, exponente en exceso a $M$ con $M = 2^{q-1}$ (donde $q$ es el número de bits del exponente) y empaquetado en hexadecimal.

# Solucionario

## Actividades

**1.** Convertid a base 10 los valores siguientes:

**a)** $10011101_{(2} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$
$= 128 + 16 + 8 + 4 + 1 = 157_{(10}$

**b)** $3AD_{(16} = 3 \cdot 16^2 + A \cdot 16^1 + D \cdot 16^0 = 3 \cdot 16^2 + 10 \cdot 16^1 + 13 \cdot 16^0 = 941_{(10}$

↑ Correspondencia del dígito A en base 10 ↑ Dígito D en base 10

**c)** $333_{(4} = 3 \cdot 4^2 + 3 \cdot 4^1 + 3 \cdot 4^0 = 48 + 12 + 3 = 63_{(10}$

**d)** $333_{(8} = 3 \cdot 8^2 + 3 \cdot 8^1 + 3 \cdot 8^0 = 192 + 24 + 3 = 219_{(10}$

**e)** $B2{,}3_{(16} = B \cdot 16^1 + 2 \cdot 16^0 + 3 \cdot 16^{-1} = 11 \cdot 16^1 + 2 \cdot 16^0 + 3 \cdot 16^{-1} =$
$= 176 + 2 + 0{,}1875 = 178{,}1875_{(10}$

**f)** $3245_{(8} = 3 \cdot 8^3 + 2 \cdot 8^2 + 4 \cdot 8^1 + 5 \cdot 8^0 = 1536 + 128 + 32 + 5 = 1701_{(10}$

**g)** $AC3C_{(16} = A \cdot 16^3 + C \cdot 16^2 + 3 \cdot 16^1 + C \cdot 16^0 = 10 \cdot 16^3 + 12 \cdot 16^2 + 3 \cdot 16^1 + 12 \cdot 16^0 =$
$= 40960 + 3072 + 48 + 12 = 44092_{(10}$

**h)** $1010{,}11_{(8} = 1 \cdot 8^3 + 0 \cdot 8^2 + 1 \cdot 8^1 + 0 \cdot 8^0 + 1 \cdot 8^{-1} + 1 \cdot 8^{-2} =$
$= 512 + 8 + 0{,}125 + 0{,}015625 = 520{,}140625_{(10}$

**i)** $110011{,}11_{(4} = 1 \cdot 4^5 + 1 \cdot 4^4 + 0 \cdot 4^3 + 0 \cdot 4^2 + 1 \cdot 4^1 + 1 \cdot 4^0 + 1 \cdot 4^{-1} + 1 \cdot 4^{-2} =$
$= 1024 + 256 + 4 + 1 + 0{,}25 + 0{,}0625 = 1285{,}3125_{(10}$

**j)** $10011001{,}1101_{(2} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 +$
$+ 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$
$= 128 + 16 + 8 + 1 + 0{,}5 + 0{,}25 + 0{,}0625 = 153{,}8125_{(10}$

**k)** $1110100{,}01101_{(2} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} +$
$+ 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} =$
$= 64 + 32 + 16 + 4 + 0{,}25 + 0{,}125 + 0{,}03125 = 116{,}40625_{(10}$

**2.** Convertid a base 2 los valores siguientes:

**a)** $425_{(10}$

Dividimos 425 por 2 sucesivamente, y registramos los restos de las divisiones enteras. Estos restos son los dígitos binarios:

| | |
|---|---|
| 425 | 1 |
| 212 | 0 |
| 106 | 0 |
| 53 | 1 |
| 26 | 0 |
| 13 | 1 |
| 6 | 0 |
| 3 | 1 |
| 1 | |

Por lo tanto: $425_{(10} = 110101001_{(2}$

**b)** $344_{(10}$

| | |
|---|---|
| 344 | 0 |
| 172 | 0 |
| 86 | 0 |
| 43 | 1 |
| 21 | 1 |
| 10 | 0 |
| 5 | 1 |
| 2 | 0 |
| 1 | |

Por lo tanto: $344_{(10} = 101011000_{(2}$

**c)** $31{,}125_{(10}$

- Parte fraccionaria

| | | | | | |
|---|---|---|---|---|---|
| 0,125 · 2 | = | 0,25 | = | 0 | + 0,25 |
| 0,25 · 2 | = | 0,50 | = | 0 | + 0,50 |
| 0,50 · 2 | = | 1,00 | = | 1 | + 0,00 |

- Parte entera

| | |
|---|---|
| 31 | 1 |
| 15 | 1 |
| 7 | 1 |
| 3 | 1 |
| 1 | |

Por lo tanto, si $0{,}125_{(10} = 0{,}001_{(2}$ i $31_{(10} = 11111_{(2}$, entonces $31{,}125_{(10} = 11111{,}001_{(2}$

**d)** $4365{,}14_{(10}$

- Parte fraccionaria

| | | | | | |
|---|---|---|---|---|---|
| 0,14 · 2 | = | 0,28 | = | 0 | + 0,28 |
| 0,28 · 2 | = | 0,56 | = | 0 | + 0,56 |
| 0,56 · 2 | = | 1,12 | = | 1 | + 0,12 |
| 0,12 · 2 | = | 0,24 | = | 0 | + 0,24 |
| 0,24 · 2 | = | 0,48 | = | 0 | + 0,48 |
| 0,48 · 2 | = | 0,96 | = | 0 | + 0,96 |
| 0,96 · 2 | = | 1,92 | = | 1 | + 0,92 |
| 0,92 · 2 | = | 1,84 | = | 1 | + 0,84 |
| 0,84 · 2 | = | 1,68 | = | 1 | + 0,68 |
| 0,68 · 2 | = | 1,36 | = | 1 | + 0,36 |
| 0,36 · 2 | = | 0,72 | = | 0 | + 0,72 |
| 0,72 · 2 | = | 1,44 | = | 1 | + 0,44 |
| ... | | | | | |

- Parte entera

| | |
|---|---|
| 4365 | 1 |
| 2182 | 0 |
| 1091 | 1 |
| 545 | 1 |
| 272 | 0 |
| 136 | 0 |
| 68 | 0 |
| 34 | 0 |
| 17 | 1 |
| 8 | 0 |
| 4 | 0 |
| 2 | 0 |
| 1 | |

Por lo tanto, si $0{,}14_{(10} = 0{,}001000111101..._{(2}$ i $4365_{(10} = 1000100001101_{(2}$, entonces

$$4365{,}14_{(10} = 4365_{(10} + 0{,}14_{(10} = 1000100001101_{(2} + 0{,}001000111101..._{(2} = \\ = 1000100001101{,}001000111101..._{(2}$$

**3.** Convertid a hexadecimal los números siguientes:

**a)** $111010011{,}1110100111_{(2}$

| **Base 2** | **000**1 | 1101 | 0011, | 1110 | 1001 | 11**00** |
|---|---|---|---|---|---|---|
| **Base 16** | 1 | D | 3, | E | 9 | C |

Por lo tanto: $111010011{,}1110100111_{(2} = 1D3{,}E9C_{(16}$

**b)** $0{,}1101101_{(2}$

| **Base 2** | 0, | 1101 | 101**0** |
|---|---|---|---|
| **Base 16** | 0, | D | A |

Por lo tanto: $0{,}1101101_{(2} = 0{,}DA_{(16}$

**c)** $45367_{(10}$

Dividiremos el valor numérico 45367 por 16 sucesivamente, y registraremos los restos de las divisiones enteras realizadas. Estos restos constituyen los dígitos hexadecimales:

| | |
|---|---|
| 45367 | 7 |
| 2835 | 3 |
| 177 | 1 |
| 11 | |

Por lo tanto: $45367_{(10} = B137_{(16}$

**d)** 111011,1010010101$_{(2}$

| **Base 2** | **00**11 | 1011, | 1010 | 0101 | 01**00** |
|---|---|---|---|---|---|
| **Base 16** | 3 | B, | A | 5 | 4 |

Por lo tanto: 111011, 1010010101 $_{(2}$ = 3B,A54$_{(16}$

**4.** Convertid los números hexadecimales siguientes a base 2, base 4 y base 8:

Podemos aprovechar la propiedad $16 = 2^4$ y $16 = 4^2$ para tratar el paso de base 16 a base 2 y a base 4 dígito a dígito. Esto es, cada dígito hexadecimal se transformará en un conjunto de cuatro dígitos binarios, mientras que cada dígito hexadecimal se puede transformar en dos dígitos en base 4.

El paso a base 8 no se puede hacer directamente desde base 16, dado que 16 no es potencia de 8. Aprovecharemos la base 2 como base intermedia. 8 es potencia de 2 ($8 = 2^3$) y, por lo tanto, tenemos una correspondencia directa: cada agrupación de tres dígitos binarios se corresponderá con un dígito octal.

**a)** ABCD$_{(16}$

| **Base 16** | A | B | C | D |
|---|---|---|---|---|
| **Base 2** | 1010 | 1011 | 1100 | 1101 |
| **Base 4** | 22 | 23 | 30 | 31 |

| **Base 2** | **00**1 | 010 | 101 | 111 | 001 | 101 |
|---|---|---|---|---|---|---|
| **Base 8** | 1 | 2 | 5 | 7 | 1 | 5 |

Por lo tanto: ABCD$_{(16}$ = 1010101111001101$_{(2}$ = 22233031$_{(4}$ = 125715$_{(8}$

**b)** 45,45$_{(16}$

| **Base 16** | 4 | 5, | 4 | 5 |
|---|---|---|---|---|
| **Base 2** | 0100 | 0101, | 0100 | 0101 |
| **Base 4** | 10 | 11, | 10 | 11 |

| **Base 2** | **0**01 | 000 | 101, | 010 | 001 | 01**0** |
|---|---|---|---|---|---|---|
| **Base 8** | 1 | 0 | 5, | 2 | 1 | 2 |

Por lo tanto: 45,45$_{(16}$ = 1000101,01000101$_{(2}$ = 1011,1011$_{(4}$ = 105,212$_{(8}$

**c)** 96FF,FF$_{(16}$

| **Base 16** | 9 | 6 | F | F, | F | F |
|---|---|---|---|---|---|---|
| **Base 2** | 1001 | 0110 | 1111 | 1111, | 1111 | 1111 |
| **Base 4** | 21 | 12 | 33 | 33, | 33 | 33 |

| **Base 2** | **00**1 | 001 | 011 | 011 | 111 | 111, | 111 | 111 | 11**0** |
|---|---|---|---|---|---|---|---|---|---|
| **Base 8** | 1 | 1 | 3 | 3 | 7 | 7, | 7 | 7 | 6 |

Por lo tanto: 96FF,FF$_{(16}$ = 1001011011111111,11111111$_{(2}$ = 21123333,3333$_{(4}$ = 113377,776$_{(8}$

**5.** Rellenad la tabla siguiente:

Como se puede ver, el valor numérico que en base 10 se representa por 74,3, con una representación en base 2, 8 y 16 tiene un número infinito de dígitos fraccionarios. En todos estos casos obtenemos una parte fraccionaria periódica.

| Binario | Octal | Hexadecimal | Decimal |
|---|---|---|---|
| **1101100,110** | 154,6 | 6C,C | 108,75 |

| Binario | Octal | Hexadecimal | Decimal |
|---|---|---|---|
| 11110010,010011 | **362,23** | F2,4C | 242,296875 |
| 10100001,00000011 | 241,006 | **A1,03** | 161,01171875 |
| 1001010,0100110011... | 112,2314631463... | 4A,4CCCCC... | **74,3** |

**6.** Empaquetad en hexadecimal la cadena de bits 10110001.

Para empaquetar sólo hay que agrupar los bits de 4 en 4 y hacer el cambio de base a hexadecimal. En este caso tenemos los grupos 1011 | 0001:

$1011_{(2} = B_{(16}$
$0001_{(2} = 1_{(16}$

Ahora, agrupamos todos estos dígitos en una única cadena y obtenemos $10110001_{(2} \rightarrow$ **B1h**.

**7.** Empaquetad en hexadecimal el número $0100000111,111010_{(2}$, que está en un formato de coma fija de 16 bits, de los cuales 6 son fraccionarios.

Se agrupan los bits de 4 en 4: 0100 | 0001 | 1111 | 1010 y se hace el cambio de base de cada grupo:

$0100_{(2} = 4_{(16}$
$0001_{(2} = 1_{(16}$
$1111_{(2} = F_{(16}$
$1010_{(2} = A_{(16}$

Por lo tanto, la cadena 0100000111111010 se empaqueta en hexadecimal por **41FAh**.

Observad que no hemos hecho un cambio de base del número fraccionario representado por la cadena de bits original, sino que hemos empaquetado los bits sin tener en cuenta su sentido.

**8.** Desempaquetad la cadena de bits A83h.

Obtenemos la codificación binaria de cada dígito:

$A_{(16} = 1010_{(2}$
$8_{(16} = 1000_{(2}$
$3_{(16} = 0011_{(2}$

Teniendo en cuenta esta correspondencia, A83h codifica la cadena de bits 101010000011.

**a)** Encontrad el valor decimal si se trata de un número natural.

Interpretado así, se trata del número binario $101010000011_{(2}$. Encontraremos el valor decimal aplicando el TFN:

$101010000011 = 2^{11} + 2^9 + 2^7 + 2^1 + 2^0 = 2048 + 512 + 128 + 2 + 1 = \mathbf{2691}_{(10}$.

**b)** Encontrad el valor decimal si se trata de un número en coma fija sin signo de 12 bits, donde 4 son fraccionarios.

Con esta interpretación, se trata del número binario $10101000,0011_{(2}$. Encontraremos el valor decimal aplicando el TFN:

$10101000,0011 = 2^7 + 2^5 + 2^3 + 2^{-3} + 2^{-4} = 128 + 32 + 8 + 0,125 + 0,06125 = \mathbf{168,1865}_{(10}$.

**9.** Consideremos el número $1010,101_{(2}$.

**a)** Haced el cambio a base 16.

Para hacer el cambio a base 16 tenemos que hacer agrupaciones de 4 bits a partir de la coma fraccionaria:

| **Base 2** | 1010, | 101**0** |
|---|---|---|
| **Base 16** | A, | A |

Por lo tanto, el número $1010,101_{(2}$ en hexadecimal es A,A (16.

**b)** Haced el empaquetamiento hexadecimal.

Para hacer el empaquetamiento, no tenemos que tener en cuenta la posición de la coma fraccionaria:

| **Base 2** | **0**101 | 0,101 |
|---|---|---|
| **Base 16** | 5 | 5 |

Por lo tanto, el número $1010{,}101_{(2}$ se empaqueta en hexadecimal como 55h.

**10.** Calculad las operaciones siguientes en la base especificada:

**a.**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | | | | | | |
| | | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | (2 |
| + | | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | (2 |
| | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | (2 |

**d.**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | (2 |
| | | | 1 | | | 1 | | | | |
| − | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | (2 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | (2 |

**b.**

| | | | | | |
|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | (8 |
| + | | 3 | 2 | 1 | (8 |
| | 2 | 6 | 6 | 6 | (8 |

**e.**

| | | | | | |
|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | (8 |
| − | | 3 | 2 | 1 | (8 |
| | 2 | 0 | 2 | 4 | (8 |

**c.**

| | | | | | |
|---|---|---|---|---|---|
| | | | 1 | | |
| | A | 2 | 3 | F | (16 |
| + | 5 | 4 | A | 3 | (16 |
| | F | 6 | E | 2 | (16 |

**f.**

| | | | | | |
|---|---|---|---|---|---|
| | A | 2 | 3 | F | (16 |
| | 1 | 1 | | | |
| − | 5 | 4 | A | 3 | (16 |
| | 4 | D | 9 | C | (16 |

**11.** Calculad las operaciones siguientes en la base especificada:

**a.**

| | | | | | |
|---|---|---|---|---|---|
| | | 1 | 1 | | |
| | 6 | 2, | 4 | 8 | (16 |
| + | 3 | 5, | D | F | (16 |
| | 9 | 8, | 2 | 7 | (16 |

**b.**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | | | | | |
| | | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1, | 1 | 1 | 0 | 1 | 1 | (2 |
| + | | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0, | 1 | 1 | 1 | | | (2 |
| | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0, | 1 | 0 | 1 | 1 | 1 | (2 |

**c.**

| | | | | | |
|---|---|---|---|---|---|
| | 6 | 2, | 4 | 8 | (16 |
| | 1 | 1 | 1 | | |
| – | 3 | 5, | D | F | (16 |
| | 2 | C, | 6 | 9 | (16 |

**d.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1, | 1 | 1 | 0 | 1 | 1 | (2 |
| | | | 1 | 1 | | | | | 1 | 1 | 1 | | | | |
| – | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0, | 1 | 1 | 1 | | | (2 |
| | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0, | 1 | 1 | 1 | 1 | 1 | (2 |

**12.** Calculad las multiplicaciones siguientes:

La multiplicación de un número por $b^k$, donde *b* es la base de numeración, equivale a desplazar la coma fraccionaria *k* posiciones a la derecha.

**a)** $128{,}7_{(10} \cdot 10^4 = 128{,}7_{(10} \cdot 10000_{(10} = 1287000_{(10}$
**b)** $AFD_{(16} \cdot 16^2 = AFD_{(16} \cdot 100_{(16} = AFD00_{(16}$
**c)** $1101{,}01_{(2} \cdot 2^2 = 1101{,}01_{(2} \cdot 100_{(2} = 110101_{(2}$

**13.** Calculad el cociente y el resto de las divisiones enteras siguientes:

La división de un número por $b^k$ donde *b* es la base de numeración, equivale a desplazar la coma fraccionaria *k* posiciones a la izquierda.

**a)** $52978_{(10} / 10^3 = 52978_{(10} / 1000_{(10} = 52{,}978_{(10}$

El cociente de la división entera es $52_{(10}$. El resto es $978_{(10}$.

**b)** $3456_{(16} / 16^2 = 3456_{(16} / 100_{(16} = 34{,}56_{(16}$.

El cociente de la división entera es $34_{(16}$. El resto es $56_{(16}$.

**c)** $100101001001_{(2} / 2^8 = 100101001001_{(2} / 100000000_{(2} = 1001{,}01001001_{(2}$

El cociente de la división entera es $1001_{(2}$. El resto es $01001001_{(2}$.

**14.** Determinad el rango y la precisión de los formatos de coma fija sin signo $x_1x_0,x_{-1}x_{-2}x_{-3}$ y , $x_2x_1x_0,x_{-1}x_{-2}$ donde $x_i$ es un dígito decimal.

Dado que la base de numeración es 10, el rango de la representación del formato $x_1x_0,x_{-1}x_{-2}x_{-3}$ es $[0, 99{,}999_{(10}]$. La precisión de este formato es $0{,}001_{(10}$ porque esta es la distancia entre dos números consecutivos representables en este formato, como por ejemplo el $12{,}121_{(10}$ y el $12{,}122_{(10}$.

De forma similar, el rango de representación del formato $x_2x_1x_0,x_{-1}x_{-2}$ es $[0, 999{,}99_{(10}]$, y su precisión es $0{,}01_{(10}$, la distancia entre dos números consecutivos representables en el formato, como por ejemplo el $45{,}77_{(10}$ y el $45{,}78_{(10}$.

**15.** Determinad si el número 925,4 se puede representar en los formatos indicados en la actividad 14.

El número $925{,}4_{(10}$ no se puede representar en el formato $x_1x_0,x_{-1}x_{-2}x_{-3}$, puesto que este número está fuera del rango de representación. En cambio, se puede representar en el formato $x_2x_1x_0,x_{-1}x_{-2}$, dado que se encuentra dentro del rango $[0, 999{,}99_{(10}]$. Además, se puede representar de manera exacta, porque en el formato hay disponibles dígitos fraccionarios suficientes.

**16.** Representad en el formato de coma fija y sin signo $x_1x_0,x_{-1}x_{-2}$, donde $x_i$ es un dígito decimal, los números siguientes:

Para escribir estos números en el formato indicado hay que escribirlos con dos dígitos enteros y dos dígitos fraccionarios:

**a)** $10_{(10}$ se escribe 10,00
**b)** $10{,}02_{(10}$ se escribe 10,02
**c)** $03{,}1_{(10}$ se escribe 03,10
**d)** $03{,}2_{(10}$ se escribe 03,20

**17.** Determinad la cantidad de números que se pueden representar en el formato $x_2x_1x_0,x_{-1}x_{-2}x_{-3}$, donde $x_i$ es un dígito decimal.

La máxima cantidad de números que se pueden representar en un formato es $b^k$, donde $k$ es el número de dígitos disponibles en el formato y $b$ la base de numeración. Dado que se trata de un formato decimal, cada dígito puede tomar 10 valores diferentes (0 - 9), y como el formato dispone de 6 dígitos para la representación, podemos representar un total de $10^6$ números. Fijémonos que la cantidad de números que se pueden representar no depende de la posición de la coma.

**18.** Calculad el error de representación que se comete cuando representamos en el formato $x_2x_1x_0,x_{-1}x_{-2}$, donde $x_i$ es un dígito decimal, los números siguientes:

**a)** $223,45_{(10}$
El número $223,45_{(10}$ está directamente representado en el formato $x_2x_1x_0,x_{-1}x_{-2}$. Por lo tanto, se trata de un número representable y el error cometido es cero.

**b)** $45,89_{(10}$
El número $45,89_{(10}$, es representable directamente en el formato $x_2x_1x_0,x_{-1}x_{-2}$. La representación es $045,89_{(10}$ y es exacta. Por lo tanto, el error de representación es cero.

**c)** $55,6356_{(10}$
El número $55,6356_{(10}$ no se puede representar directamente en el formato $x_2x_1x_0,x_{-1}x_{-2}$, dado que tiene 4 dígitos fraccionarios. Tendremos que hacer una aproximación, lo que comporta un cierto error de aproximación.

Con una aproximación por truncamiento, la representación será $55,635_{(10}$. El error de representación que se comete es $|55,6356_{(10} - 55,635_{(10}| = 0,0006_{(10}$.

Para encontrar la representación una aproximación por redondeo, tenemos que sumar la mitad de la precisión y truncar el resultado a 3 dígitos fraccionarios: $55,6356_{(10} + 0,001_{(10} = 55,6366_{(10}$ que con el truncamiento a 3 dígitos fraccionarios queda $55,636_{(10}$. El error de representación cometido es, en este caso, $|55,6356_{(10} - 55,636_{(10}| = 0,0004_{(10}$.

**d)** $23,56_{(10}$
El número $23,56_{(10}$, es representable directamente en el formato $x_2x_1x_0,x_{-1}x_{-2}$. La representación es $023,56_{(10}$ y es exacta. Por lo tanto, el error de representación es cero.

**19.** Escoged el formato hexadecimal que use el mínimo número de dígitos y que permita representar el número $16,25_{(10}$ de manera exacta. ¿Cuál es el rango y la precisión del formato?

Para representar este número en hexadecimal, primero lo pasaremos a binario. La representación binaria de $16,25_{(10}$ es $10000,01_{(2}$.

Para la representación hexadecimal de este número hay que añadir ceros a los extremos hasta disponer de grupos de 4 bits completos a ambos lados de la coma decimal. Entonces tenemos:

$$00010000,0100_{(2} = 10,4_{(16}.$$

- El rango de esta representación es:[0 ($00,0_{(16}$ ), $255,9375_{(10}$ (FF,$F_{(16)}$)].
- Su precisión es: $0,0625_{(10} = 00,1_{(16} - 00,0_{(16}$.

**20.** ¿Cuál es el número más pequeño que hay que sumar a $8341_{(10}$ para que se produzca desbordamiento en una representación decimal (base 10) de cuatro dígitos?

Para que en una representación decimal de 4 dígitos sin signo se produzca desbordamiento, tenemos que sobrepasar el número $9999_{(10}$; por lo tanto, tendremos desbordamiento cuando la suma de dos números sea $10000_{(10}$. Entonces, el número más pequeño que tenemos que sumar a $8341_{(10}$ es $10000_{(10} - 8341_{(10} = 1659_{(10}$.

**21.** Convertid los valores decimales siguientes a binarios en los sistemas de representación de signo y magnitud y complemento a 2, con un formato entero de 8 bits:

Pasamos los valores numéricos a binario:

**a) 53**

| | |
|---|---|
| 53 | 1 |
| 26 | 0 |
| 13 | 1 |
| 6 | 0 |
| 3 | 1 |
| 1 | |

$+53_{(10} = +110101_{(2}$

**b) –25**

| | |
|---|---|
| 25 | 1 |
| 12 | 0 |
| 6 | 0 |
| 3 | 1 |
| 1 | |

$-25_{(10} = -11001_{(2}$

**c) 93**

| | |
|---|---|
| 93 | 1 |
| 46 | 0 |
| 23 | 1 |
| 11 | 1 |
| 5 | 1 |
| 2 | 0 |
| 1 | |

$+93_{(10} = +1011101_{(2}$

**d)** –1

| 1 | 1 |
|---|---|
| 0 | |

**e)** –127

| 127 | 1 |
|---|---|
| 63 | 1 |
| 31 | 1 |
| 15 | 1 |
| 7 | 1 |
| 3 | 1 |
| 1 | |

**f)** –64

| 64 | 0 |
|---|---|
| 32 | 0 |
| 16 | 0 |
| 8 | 0 |
| 4 | 0 |
| 2 | 0 |
| 1 | |

$-1_{(10} = -1_{(2}$   $-127_{(10} = -1111111_{(2}$   $-64_{(10} = -1000000_{(2}$

Para obtener la representación en signo y magnitud, tan sólo tenemos que poner el bit de signo y añadir la magnitud expresada en 7 bits:

| Base 10 | Base 2 | Signo y magnitud |
|---|---|---|
| $+53_{(10}$ | $+110101_{(2}$ | $00110101_{(SM2}$ |
| $-25_{(10}$ | $-11001_{(2}$ | $10011001_{(SM2}$ |
| $+93_{(10}$ | $+1011101_{(2}$ | $01011101_{(SM2}$ |
| $-1_{(10}$ | $-1_{(2}$ | $10000001_{(SM2}$ |
| $-127_{(10}$ | $-1111111_{(2}$ | $11111111_{(SM2}$ |
| $-64_{(10}$ | $-1000000_{(2}$ | $11000000_{(SM2}$ |

La representación en Ca2 de las magnitudes positivas coincide con la representación en signo y magnitud. La representación en Ca2 de las magnitudes negativas se puede obtener de varias maneras:

- Se puede hacer la operación $2^8 - |X|$ en base 10, y pasar posteriormente el resultado a binario.
- Podemos hacer la operación $2^8 - |X|$ directamente en base 2.
- Se aplica un cambio de signo a la magnitud positiva en Ca2.

**a)** El $+53_{(10} = +110101_{(2}$ se representa por $00110101_{(Ca2}$ en Ca2.

**b)** Podemos obtener la representación en Ca2 del $-25_{(10}$ de la manera siguiente:

- $2^8 - 25 = 256_{(10} - 25_{(10} = 231_{(10} = 11100111_{(Ca2}$, o bien,
- $2^8 - 25 = 100000000_{(2} - 11001_{(2} = 11100111_{(Ca2}$, o bien,
- $+25_{(10} = +11001_{(2}$ → Representación de la magnitud positiva → $00011001_{(Ca2}$ →
  → Cambio de signo → $11100111_{(Ca2}$

**c)** El $+93_{(10} = +1011101_{(2}$ se representa por $01011101_{(Ca2}$ en Ca2.

**d)** Obtenemos la representación en Ca2 del $-1_{(10}$:

- $2^8 - 1 = 256_{(10} - 1_{(10} = 255_{(10} = 11111111_{(Ca2}$, o bien,
- $2^8 - 1 = 100000000_{(2} - 1_{(2} = 11111111_{(Ca2}$, o bien,
- $+1_{(10} = +1_{(2}$ → Representación de la magnitud positiva → $00000001_{(Ca2}$ →
  → Cambio de signo → $11111111_{(Ca2}$

**e)** La representación en Ca2 del $-127_{(10}$ se puede obtener:

- $2^8 - 127 = 256_{(10} - 127_{(10} = 129_{(10} = 10000001_{(Ca2}$, o bien,
- $2^8 - 127 = 100000000_{(2} - 1111111_{(2} = 10000001_{(Ca2}$, o bien,
- $+127_{(10} = +1111111_{(2}$ → Representación de la magnitud positiva → $01111111_{(Ca2}$ →
  → Cambio de signo → $10000001_{(Ca2}$

**f)** La representación en Ca2 del $-64_{(10}$ se obtiene:

- $2^8 - 64 = 256_{(10} - 64_{(10} = 192_{(10} = 11000000_{(Ca2}$, o bien,
- $2^8 - 64 = 100000000_{(2} - 1000000_{(2} = 11000000_{(Ca2}$, o bien,
- $+64_{(10} = +1000000_{(2}$ → Representación de la magnitud positiva → $01000000_{(Ca2}$ →
  → Cambio de signo → $11000000_{(Ca2}$

| Decimal | Binario | Complemento a 2 | Signo y magnitud |
|---|---|---|---|
| $+53_{(10}$ | $+110101_{(2}$ | $00110101_{(Ca2}$ | $00110101_{(SM2}$ |
| $-25_{(10}$ | $-11001_{(2}$ | $11100111_{(Ca2}$ | $10011001_{(SM2}$ |
| $+93_{(10}$ | $+1011101_{(2}$ | $01011101_{(Ca2}$ | $01011101_{(SM2}$ |
| $-1_{(10}$ | $-1_{(2}$ | $11111111_{(Ca2}$ | $10000001_{(SM2}$ |
| $-127_{(10}$ | $-1111111_{(2}$ | $10000001_{(Ca2}$ | $11111111_{(SM2}$ |
| $-64_{(10}$ | $-1000000_{(2}$ | $11000000_{(Ca2}$ | $11000000_{(SM2}$ |

**22.** Si tenemos los números binarios 00110110, 11011010, 01110110, 11111111 y 11100100, ¿cuáles son los equivalentes decimales considerando que son valores binarios representados en signo y magnitud?

Si consideramos que son valores en signo y magnitud, tenemos:

$00110110_{(SM2} = +54_{(10}$
$11011010_{(SM2} = -90_{(10}$
$01110110_{(SM2} = +118_{(10}$
$11111111_{(SM2} = -127_{(10}$
$11100100_{(SM2} = -100_{(10}$

**23.** Repetid el ejercicio anterior considerando que las cadenas de bits son números en complemento a 2.

Si consideramos que son valores en complemento a 2:

$00110110_{(Ca2} = +54_{(10}$
$11011010_{(Ca2} = -2^7+ 90_{(10} = -38_{(10}$
$01110110_{(Ca2} = +118_{(10}$
$11111111_{(Ca2} = -2^7 + 127_{(10} = -1_{(10}$
$11100100_{(Ca2} = -2^7 + 100_{(10} = -28_{(10}$

**24.** Si tenemos las cadenas de bits siguientes $A = 1100100111$, $B = 1000011101$ y $C = 0101011011$, haced las operaciones que proponemos a continuación considerando que son números binarios en formato de signo y magnitud: $A + B$, $A - B$, $A + C$, $A - C$, $B - C$, $B + C$.

Para hacer las operaciones de suma y resta en signo y magnitud, tenemos que tener en cuenta el signo de las magnitudes, y actuar consecuentemente. Para lograr más claridad, junto a la operación en binario se ha hecho el equivalente en base 10. Los valores $A$ , $B$ y C en base 10 corresponden a:

$A = 1100100111 = -295_{(10}$
$B = 1000011101 = -29_{(10}$
$C = 0101011011 = +347_{(10}$

- **Suma $A + B$**

*A es* negativo y *B* también. Por lo tanto, haremos la suma de las magnitudes (sin signo) y si no hay desbordamiento añadiremos un 1 como bit de signo al resultado obtenido, para obtener el resultado en la representación en signo y magnitud:

**Suma $A + B$**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 1 | 1 | 1 | 1 | 1 | | | | | 1 | 1 | | |
| | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (SM2 | | | 2 | 9 | 5 | (10 |
| + | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (SM2 | + | | | 2 | 9 | (10 |
| | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | (SM2 | | | 3 | 2 | 4 | (10 |

$A + B = 1101000100_{(2} = -324_{(10}$

- **Resta *A* – *B***

*A es* negativo y *B* también. Por lo tanto, restaremos la magnitud pequeña (*B*) de la magnitud grande (*A*) y aplicaremos el signo de la magnitud grande (*A*) al resultado (no se puede producir desbordamiento):

**Resta *A* – *B***

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (SM2 | | 2 | 9 | 5 | (10 |
| | | | | 1 | 1 | | | | | | | | 1 | | |
| – | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (SM2 | – | | 2 | 9 | (10 |
| | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | (SM2 | | 2 | 6 | 6 | (10 |

$A - B = 1100001010_{(2} = -266_{(10}$

- **Suma *A* + *C***

*A es* negativo y *C* es positivo. Por lo tanto, restaremos la magnitud pequeña (*A)* de la magnitud grande (*C*) y aplicaremos el signo de la magnitud grande (*C*) al resultado (no se puede producir desbordamiento):

**Suma *A* + *C***

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | (SM2 | | 3 | 4 | 7 | (10 |
| | | | 1 | | | 1 | | | | | | 1 | | | |
| – | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (SM2 | – | 2 | 9 | 5 | (10 |
| | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | (SM2 | | 0 | 5 | 2 | (10 |

$A + C = 0000110100_{(2} = +52_{(10}$

- **Resta *A* – *C***

*A es* negativo y *C* es positivo. Por lo tanto, haremos la suma de las magnitudes (sin signo) y, si no hay desbordamiento, añadiremos un 1 como bit de signo al resultado obtenido, para obtener el resultado en la representación en signo y magnitud:

**Resta *A* – *C***

Desbordamiento
↓

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | 1 | 1 | | |
| | | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (SM2 | | 2 | 9 | 5 | (10 |
| + | | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | (SM2 | + | 3 | 4 | 7 | (10 |
| | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | (SM2 | | 6 | 4 | 2 | (10 |

$A - C = -642_{(10}$
No se puede representar con 10 bits
en el formato de signo y magnitud.

- **Resta *B* − *C***

*B* es negativo y *C* es positivo. Por lo tanto, haremos la suma de las magnitudes (sin signo) y, si no hay desbordamiento, añadiremos un 1 como bit de signo al resultado obtenido, para obtener el resultado en la representación en signo y magnitud:

**Resta *B* − *C***

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 1 | 1 | 1 | | |
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (SM2 |
| + | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | (SM2 |
| | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | (SM2 |

| | | | | |
|---|---|---|---|---|
| | | 1 | | |
| | | 2 | 9 | (10 |
| + | 3 | 4 | 7 | (10 |
| | 3 | 7 | 6 | (10 |

$B - C = 1101111000_{(2} = -376_{(10}$

- **Suma *B* + *C***

*B* es negativo y *C* es positivo. Por lo tanto, restaremos la magnitud pequeña (*B*) de la magnitud grande (*C*) y aplicaremos el signo de la magnitud más grande (*C*) al resultado (no se puede producir desbordamiento):

**Suma *B* + *C***

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | (SM2 |
| | | | 1 | 1 | 1 | 1 | | | | |
| − | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (SM2 |
| | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | (SM2 |

| | | | | |
|---|---|---|---|---|
| | 3 | 4 | 7 | (10 |
| | | 1 | | |
| − | | 2 | 9 | (10 |
| | 3 | 1 | 8 | (10 |

$B + C = 0100111110_{(2} = +318_{(10}$

**25.** Repetid la actividad anterior considerando que las cadenas representan números en complemento a 2.
Para hacer las operaciones de suma en Ca2, operaremos directamente sobre las representaciones. El resultado será correcto siempre que no se produzca desbordamiento. Para hacer las operaciones de resta, aplicaremos un cambio de signo al sustraendo y haremos una operación de suma:

$$A = 1100100111 = -217_{(10}$$
$$B = 1000011101 = -483_{(10}$$
$$C = 0101011011 = +347_{(10}$$

- **Suma *A* + *B***

**Suma *A* + *B***

Desbordamiento
↓

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | 1 | 1 | 1 | 1 | 1 | 1 | | |
| | | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (Ca2 |
| + | | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (Ca2 |
| | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | (Ca2 |

| | | | | |
|---|---|---|---|---|
| | 1 | 1 | | |
| | −2 | 1 | 7 | (10 |
| + | −4 | 8 | 3 | (10 |
| | −7 | 0 | 0 | (10 |

$A + B = -700_{(10}$

Se produce desbordamiento, dado que al sumar dos números negativos obtenemos uno positivo. El resultado no cabe en el formato de salida. Recordemos que el rango de representación de enteros con 10 bits con el formato de complemento a 2 es $[-2^{10-1}, 2^{10-1} - 1] = [-512, 511]$

- **Resta** $A - B$

Aplicaremos un cambio de signo a $B$ y haremos una operación de suma:

```
1000011101
          ^ Se mantienen los bits hasta aquí (el primer 1 que encontramos, incluyendo éste)
          1
         ^ Se complementan los bits a partir de este punto
011110001
```

Al aplicar un cambio de signo al 1000011101 obtenemos el 0111100011, que será el valor que utilizaremos en la suma:

**Suma** $A + (-B)$

```
  Transporte que se ignora
  ↓
  1 1 1 1 1     1 1 1                          1
    1 1 0 0 1 0 0 1 1 1 (Ca2          – 2  1  7 (10
+   0 1 1 1 1 0 0 0 1 1 (Ca2      +   + 4  8  3 (10
  ---------------------           -------------
  1 0 1 0 0 0 0 1 0 1 0 (Ca2          + 2  6  6 (10
```

$A - B = 0100001010 = +266_{(10}$

- **Suma** $A + C$

$A$ es negativo y $C$ es positivo. No se puede producir desbordamiento:

**Suma** $A + C$

```
  Transporte que se ignora
  ↓
  1 1   1 1 1 1 1 1 1
    1 1 0 0 1 0 0 1 1 1 (Ca2          – 2  1  7 (10
+   0 1 0 1 0 1 1 0 1 1 (Ca2      +   + 3  4  7 (10
  ---------------------           -------------
  1 0 0 1 0 0 0 0 0 1 0 (Ca2          + 1  3  0 (10
```

$A + C = 0010000010 = +130_{(10}$

- **Resta** $A - C$

Aplicaremos un cambio de signo a $C$ y haremos una operación de suma:

```
0101011011   ←   Valor numérico inicial
1010100100   ←   Complemento bit a bit de la expresión inicial
+        1   ←   Sumamos 1 al bit menos significativo del formato
----------
1010100101
```

Al aplicar un cambio de signo al 0101011011 obtenemos el 1010100101, que será el valor que utilizaremos en la suma:

**Suma $A + (-C)$**

Desbordamiento
↓

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | 1 | | | 1 | 1 | 1 | | | | | 1 | | |
| | | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | (Ca2 | | –2 | 1 | 7 | (10 |
| + | | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | (Ca2 | + | –3 | 4 | 7 | (10 |
| | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | (Ca2 | | –5 | 6 | 4 | (10 |

$A - C = -564_{(10}$

Se produce desbordamiento, dado que al sumar dos números negativos, obtenemos uno positivo. El resultado no cabe en el formato de salida. Recordemos que el rango de representación de enteros con 10 bits con el formato de complemento a 2 es $[-2^{10-1}, 2^{10-1} - 1] = [-512, 511]$.

- **Resta $B - C$**

Aplicaremos un cambio de signo a $C$ y haremos una operación de suma. Al aplicar un cambio de signo al 0101011011 obtenemos el 1010100101, que será el valor que utilizaremos en la suma:

**Suma $B + (-C)$**

Desbordamiento
↓

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | 1 | 1 | 1 | 1 | | 1 | | | | 1 | 1 | | |
| | | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (Ca2 | | –4 | 8 | 3 | (10 |
| + | | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | (Ca2 | + | –3 | 4 | 7 | (10 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | (Ca2 | | –8 | 3 | 0 | (10 |

$B - C = -830_{(10}$

Se produce desbordamiento, dado que al sumar dos números negativos, obtenemos uno positivo. El resultado no cabe en el formato de salida.

- **Suma $B + C$**

$B$ es negativo y $C$ es positivo. No se puede producir desbordamiento:

**Suma $B + C$**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | 1 | 1 | 1 | 1 | | | | | 1 | | |
| | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | (Ca2 | | –4 | 8 | 3 | (10 |
| + | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | (Ca2 | + | +3 | 4 | 7 | (10 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | (Ca2 | | –1 | 3 | 6 | (10 |

$B + C = 1101111000 = -136_{(10}$

**26.** Si tenemos la cadena de bits 10110101, haced las conversiones siguientes:

**a)** Considerando que representa un número en Ca2, representad el mismo número en signo y magnitud y 16 bits:

Si la cadena de bits 10110101 representa un número en Ca2, se trata de un número negativo, puesto que el primer bit es 1. Podemos hacer un cambio de signo para obtener la magnitud positiva:

```
  | 1 0 1 1 0 1 0 1 ← Valor numérico inicial
  |
  | 0 1 0 0 1 0 1 0 ← Complemento bit a bit de la expresión inicial
+ |               1 ← Sumamos 1 al bit menos significativo del formato
-------------------
  | 0 1 0 0 1 0 1 1
```

Para codificar el valor inicial en signo y magnitud, sólo tenemos que cambiar el bit de signo del valor conseguido con el cambio de signo. Por lo tanto, la codificación en signo y magnitud del valor 10110101 que está en Ca2 es $\mathbf{11001011}_{(SM2}$.

La codificación en 16 bits la podemos obtener haciendo la extensión del formato, que en este caso se consigue añadiendo los ceros necesarios a la derecha del signo:

$$11001011_{(SM2} = \mathbf{1000000001001011}_{(SM2}$$

**b)** Considerando que representa un número en signo y magnitud, representad el mismo número en Ca2 y 16 bits:

Si se trata de un número codificado en signo y magnitud, es un número negativo, puesto que el primer dígito corresponde al signo y es 1. El resto de dígitos codifican la magnitud en binario. Podemos obtener la magnitud positiva cambiando el bit de signo: $00110101_{(2}$.

La representación de los valores positivos coincide en Ca2 y en signo y magnitud. Por lo tanto, podemos interpretar que tenemos la magnitud positiva en Ca2 y que podemos conseguir la magnitud negativa en Ca2 aplicando un cambio de signo:

```
  | 0 0 1 1 0 1 0 1 ← Valor numérico inicial
  |
  | 1 1 0 0 1 0 1 0 ← Complemento bit a bit de la expresión inicial
+ |               1 ← Sumamos 1 al bit menos significativo del formato
-------------------
  | 1 1 0 0 1 0 1 1
```

Por lo tanto, la codificación en Ca2 del valor $10110101_{(SM2}$ que está en signo y magnitud es $\mathbf{11001011}_{(Ca2}$.

La codificación en 16 bits la podemos obtener haciendo la extensión del formato, que en este caso se consigue copiando el bit de signo a la izquierda de la codificación tantas veces como sea necesario:

$$11001011_{(Ca2} = \mathbf{1111111111001011}_{(Ca2}$$

**27.** Determinad qué valor decimal codifica la cadena de bits 1010010 en los supuestos siguientes:

**a)** Si se trata de un número en coma fija sin signo de 7 bits donde 4 son fraccionarios.

Con este formato, el número binario que representa esta tira de bits es $101{,}0010_{(2}$ y, al aplicar el TFN, se obtiene el número decimal que representa:

$$\begin{aligned} 101{,}0010_{(2} &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} \\ &= 2^2 + 2^0 + 2^{-3} \\ &= 4 + 1 + 0{,}125 \\ &= \mathbf{5{,}125}_{(10} \end{aligned}$$

**b)** Si se trata de un número en coma fija sin signo de 7 bits donde 1 es fraccionario.

En este caso, el número binario que representa esta tira de bits es $101001{,}0_{(2}$ y, al aplicar el TFN, se obtiene el número decimal que representa:

$$
\begin{aligned}
101001{,}0_{(2} &= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} \\
&= 2^5 + 2^3 + 2^0 \\
&= 32 + 8 + 1 \\
&= \mathbf{41}_{(10}
\end{aligned}
$$

**28.** Codificad los números $+12{,}85_{(10}$, $+0{,}7578125_{(10}$ y $-11{,}025_{(10}$ en una representación fraccionaria binaria en signo y magnitud de 8 bits donde 3 son fraccionarios. Usad una aproximación por redondeo en caso de que sea necesario.

Codificamos el número $+12{,}85_{(10}$ en el formato especificado. Como se trata de un número positivo, el bit de signo es **0**. En cuanto a la magnitud $12{,}85_{(10}$, primero pasamos a binario la parte entera y, posteriormente, la parte fraccionaria. Para la parte entera usamos el algoritmo de divisiones sucesivas por la base de llegada (2):

$$
\begin{array}{rclcl}
12 &=& 6 \cdot 2 &+& \mathbf{0} \\
6 &=& 3 \cdot 2 &+& \mathbf{0} \\
3 &=& 1 \cdot 2 &+& \mathbf{1} \\
1 &=& 0 \cdot 2 &+& \mathbf{1}
\end{array} \uparrow
$$

Así pues, $12_{(10} = 1100_{(2}$. En cuanto a la parte fraccionaria, hacemos multiplicaciones sucesivas por la base de llegada (2):

$$
\begin{array}{lclcl}
0{,}85 \cdot 2 &=& 1{,}70 &=& \mathbf{1} + 0{,}70 \\
0{,}70 \cdot 2 &=& 1{,}40 &=& \mathbf{1} + 0{,}40 \\
0{,}40 \cdot 2 &=& 0{,}80 &=& \mathbf{0} + 0{,}80 \\
0{,}80 \cdot 2 &=& 1{,}60 &=& \mathbf{1} + 0{,}60 \\
0{,}60 \cdot 2 &=& 1{,}20 &=& \mathbf{1} + 0{,}20
\end{array} \downarrow
$$

Como el formato especificado sólo tiene 3 bits para la parte fraccionaria, ya tenemos más bits de los necesarios y podemos parar el proceso aquí. Por lo tanto, $0{,}85_{(10} = 0{,}11011\ldots_{(2}$. Para aproximar el valor con 3 bits y redondeo, procedemos como sigue:

$$0{,}11011_{(2} + 0{,}0001_{(2} = 0{,}11101_{(2} \rightarrow \text{truncamos 3 bits} \rightarrow 0{,}111_{(2}$$

A continuación, juntamos las partes entera y fraccionaria y obtenemos: $12{,}85_{(10} \approx 1100{,}111_{(2}$. Finalmente, para obtener la representación en el formato indicado, hay que añadir el bit de signo, de forma que la tira de bits que representa este número en el formato dado es: $\mathbf{0}1100{,}111_{(2}$. Hay que recordar que la tira de bits que se almacenaría en un computador no contiene la coma ni la especificación de la base: **01100111**.

En los otros dos casos, procedemos de manera totalmente análoga:

$+0{,}7578125_{(10}$:

- El bit de signo es 0, porque el número es positivo.
- La parte entera es $0_{(10} = 0_{(2}$. Como tenemos 4 bits para la parte entera, escribiremos $0000_{(2}$.
- En cuanto a la parte fraccionaria:

$$
\begin{array}{lclcl}
0{,}7578125 \cdot 2 &=& 1{,}515625 &=& \mathbf{1} + 0{,}515625 \\
0{,}515625 \cdot 2 &=& 1{,}03125 &=& \mathbf{1} + 0{,}03125 \\
0{,}03125 \cdot 2 &=& 0{,}0625 &=& \mathbf{0} + 0{,}0625 \\
0{,}0625 \cdot 2 &=& 0{,}125 &=& \mathbf{0} + 0{,}125 \\
0{,}1250 \cdot 2 &=& 0{,}25 &=& \mathbf{0} + 0{,}25
\end{array} \downarrow
$$

Así, $0{,}7578125_{(10} = 0{,}11000\ldots_{(2}$ que, redondeando con 3 bits:

$$0{,}11000_{(2} + 0{,}0001_{(2} = 0{,}11010_{(2} \rightarrow \text{truncamos 3 bits} \rightarrow 0{,}110_{(2}$$

Finalmente, se juntan todas las partes y obtenemos $0{,}7578125_{(10} \approx \mathbf{0}0000{,}110_{(2}$. Por lo tanto, la tira de bits que se almacenaría en un computador es **00000110**.

$-11{,}025_{(10}$:

- El bit de signo es 1 porque el número es negativo.

- La parte entera es $11_{(10}$:

$$\begin{array}{rcrcl} 11 & = & 5 \cdot 2 & + & \mathbf{1} \\ 5 & = & 2 \cdot 2 & + & \mathbf{1} \\ 2 & = & 1 \cdot 2 & + & \mathbf{0} \\ 1 & = & 0 \cdot 2 & + & \mathbf{1} \end{array} \uparrow$$

Es decir$11_{(10} = 1011_{(2}$.

- En cuanto a la parte fraccionaria:

$$\begin{array}{lclcl} 0{,}025 \cdot 2 & = & 0{,}05 & = & \mathbf{0} + 0{,}05 \\ 0{,}05 \cdot 2 & = & 0{,}1 & = & \mathbf{0} + 0{,}1 \\ 0{,}1 \cdot 2 & = & 0{,}2 & = & \mathbf{0} + 0{,}2 \\ 0{,}2 \cdot 2 & = & 0{,}4 & = & \mathbf{0} + 0{,}4 \\ 0{,}4 \cdot 2 & = & 0{,}8 & = & \mathbf{0} + 0{,}8 \end{array} \downarrow$$

Así, $0{,}025_{(10} = 0{,}00000..._{(2}$ que, redondeando con 3 bits:

$$0{,}00000_{(2} + 0{,}0001_{(2} = 0{,}00010_{(2} \rightarrow \text{truncamos 3 bits} \rightarrow 0{,}000_{(2}$$

Finalmente, se juntan todas las partes y obtenemos $-11{,}025_{(10} \approx \mathbf{11011{,}000}_{(2}$. Por lo tanto, la tira de bits que se almacenaría en un computador es **11011000**.

**29.** Si tenemos una representación en coma fija en signo y magnitud binaria de 8 bits, donde 3 bits son fraccionarios, determinad los números codificados por las cadenas de bits 01001111, 11001111, 01010100, 00000000 y 10000000.

Para obtener la representación decimal de estas cadenas de bits, sólo hay que aplicar el TFN considerando que los tres últimos bits constituyen la parte fraccionaria y que el primer bit representa el signo:

$01001111 \rightarrow +1001{,}111_{(2}$

$$\begin{aligned} +1001{,}111_{(2} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 2^3 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} \\ &= 8 + 1 + 0{,}5 + 0{,}25 + 0{,}125 \\ &= \mathbf{+9{,}875}_{(10} \end{aligned}$$

$11001111 \rightarrow -1001{,}111_{(2}$

$$\begin{aligned} -1001{,}111_{(2} &= -(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}) \\ &= -(2^3 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3}) \\ &= -(8 + 1 + 0{,}5 + 0{,}25 + 0{,}125) \\ &= \mathbf{-9{,}875}_{(10} \end{aligned}$$

$01010100 \rightarrow +1010{,}100_{(2}$

$$\begin{aligned} +1010{,}100_{(2} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} \\ &= 2^3 + 2^1 + 2^{-1} \\ &= 8 + 2 + 0{,}5 \\ &= \mathbf{+10{,}5}_{(10} \end{aligned}$$

$00000000 \rightarrow +0000{,}000_{(2}$

$$\begin{aligned} +0000{,}000_{(2} &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} \\ &= \mathbf{+0}_{(10} \end{aligned}$$

$10000000 \rightarrow -0000{,}000_{(2}$

$$\begin{aligned} -0000{,}000_{(2} &= -(0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3}) \\ &= \mathbf{-0}_{(10} \end{aligned}$$

**30.** Si las cadenas de bits 00101010, 11010010 y 10100010 representan números en coma fija sin signo de 8 bits, donde 3 son fraccionarios, representadlos en un formato de coma fija sin signo de 12 bits donde 4 son fraccionarios.

Para extender el rango de estas tiras de bits hay que añadir ceros tanto a la izquierda como a la derecha de la magnitud, dado que se trata de magnitudes sin signo.

La representación original tiene 5 bits para la parte entera y la representación de llegada tiene 8. Por lo tanto, hay que añadir 8 – 5 = 3 bits a la izquierda de la magnitud. Del mismo modo, la representación original tiene 3 bits para la parte fraccionaria, mientras que la representación de llegada tiene 4. Así pues, hay que añadir 4 – 3 = 1 bits a la derecha de la magnitud:

00101010 → 00101,010 → **000**00101,010**0** → 000001010100
11010010 → 11010,010 → **000**11010,010**0** → 000110100100
10100010 → 10100,010 → **000**10100,010**0** → 000101000100

**31.** Repetid la actividad anterior considerando que se trata de números en signo y magnitud.

En este caso, hay que considerar que el primer bit representa el signo. La extensión de la parte entera implica replicar el bit de signo e insertar tantos ceros como se convenga. En este caso pasamos de 4 a 7 bits para la parte entera, por lo tanto, habrá que añadir 7 – 4 = 3 bits a la izquierda de la parte entera. En cuanto a la parte fraccionaria, la situación es idéntica al ejercicio anterior. Así pues, habrá que añadir 4 – 3 = 1 bit a la derecha de la magnitud:

00101010 → +0101,010 → **+000**0101,010**0** → 000001010100
11010010 → –1010,010 → **–000**1010,010**0** → 100010100100
10100010 → –0100,010 → **–000**0100,010**0** → 100001000100

**32.** Determinad el rango de representación y la precisión en los formatos siguientes:

**a)** Coma fija en signo y magnitud con 8 bits, donde 3 son fraccionarios.

El rango de una representación fraccionaria con signo y magnitud es:

$$[-2^{n-m-1} + 2^{-m}, +2^{n-m-1} - 2^{-m}]$$

donde $n$ es el número de bits empleado en la representación y $m$ el número dígitos fraccionarios. Por lo tanto, el rango en este caso se obtiene haciendo $n = 8$ y $m = 3$, es decir:

$$[-2^{8-3-1} + 2^{-3}, +2^{8-3-1} - 2^{-3}] = [-2^4 + 2^{-3}, +2^4 - 2^{-3}] = \mathbf{[-15{,}875_{(10}, +15{,}875_{(10}]}$$

En cuanto a la precisión, ésta viene dada por el bit de menor peso de la magnitud, concretamente es igual a $2^{-m} = 2^{-3} = \mathbf{0{,}125_{(10}}$.

**b)** Coma fija en signo y magnitud con 8 bits, donde 4 son fraccionarios.

En este caso, la situación es la misma que en el apartado anterior pero con $n = 8$ y $m = 4$. El rango es el siguiente:

$$[-2^{8-4-1} + 2^{-4}, +2^{8-4-1} - 2^{-4}] = [-2^3 + 2^{-4}, +2^3 - 2^{-4}] = \mathbf{[-7{,}9375_{(10}, +7{,}9375_{(10}]}$$

Análogamente, la precisión es igual a $2^{-m} = 2^{-4} = \mathbf{0{,}0625_{(10}}$.

**c)** Coma fija sin signo con 8 bits, donde 3 son fraccionarios.

En coma fija sin signo, el rango de representación viene dado por:

$$[0, +2^{n-m} - 2^{-m}]$$

donde $n$ es el número de bits empleado en la representación y $m$ el número dígitos fraccionarios. Por lo tanto, el rango en este caso se obtiene haciendo $n = 8$ y $m = 3$, es decir:

$$[0, +2^{8-3} - 2^{-3}] = [0, +2^5 - 2^{-3}] = \mathbf{[0, +31{,}875_{(10}]}$$

La precisión es la misma que en los casos anteriores, porque es independiente de si la representación es sin signo o con signo y magnitud. Por lo tanto, la precisión es $2^{-m} = 2^{-3} = 0{,}125$.

**d)** Coma fija sin signo con 8 bits, donde 4 son fraccionarios.

En este caso, la situación es la misma que en el apartado anterior pero con $n = 8$ y $m = 4$. El rango es el siguiente:

$$[0, +2^{8-4} - 2^{-4}] = [0, +2^4 - 2^{-4}] = \mathbf{[0, +15{,}9375_{(10}]}$$

Análogamente, la precisión es $2^{-m} = 2^{-4} = \mathbf{0{,}0625_{(10}}$.

**33.** Determinad la precisión necesaria para poder representar el número $+0{,}1875_{(10}$ de forma exacta (sin error de representación) con un formato de coma fija en base 2.

Para determinar la precisión necesaria para representar el número $+0{,}1875_{(10}$ de forma exacta, primero codificaremos en binario este número haciendo multiplicaciones sucesivas por la base de llegada:

$$
\begin{array}{llllll}
0{,}1875 \cdot 2 & = & 0{,}375 & = & \mathbf{0} + 0{,}375 & \downarrow \\
0{,}375 \cdot 2 & = & 0{,}75 & = & \mathbf{0} + 0{,}75 & \\
0{,}75 \cdot 2 & = & 1{,}5 & = & \mathbf{1} + 0{,}5 & \\
0{,}5 \cdot 2 & = & 1 & = & \mathbf{1} + 0 &
\end{array}
$$

Por lo tanto, $0{,}1875_{(10}=0{,}0011_{(2}$. Así pues, para poder representar este número de forma exacta, es necesario que la representación disponga, como mínimo, de 4 dígitos fraccionarios ($m$ = 4) y, por lo tanto, una precisión de la forma $2^{-m} = 2^{-4} = \mathbf{0{,}0625_{(10}}$.

**34.** Determinad las características de rango y precisión, así como el número de dígitos enteros y fraccionarios necesarios en un formato de coma fija en signo y magnitud, para poder representar de forma exacta los números $+31{,}875_{(10}$ y $-16{,}21875_{(10}$.

En primer lugar, obtendremos la codificación binaria de estos dos números.

Por el $+31{,}875_{(10}$:

- El bit de signo es 0, porque el número es positivo.
- La parte entera es $31_{(10}$:

$$
\begin{array}{llllllll}
31 & = & 15 & \cdot & 2 & + & \mathbf{1} & \uparrow \\
15 & = & 7 & \cdot & 2 & + & \mathbf{1} & \\
7 & = & 3 & \cdot & 2 & + & \mathbf{1} & \\
3 & = & 1 & \cdot & 2 & + & \mathbf{1} & \\
1 & = & 0 & \cdot & 2 & + & \mathbf{1} &
\end{array}
$$

Es decir $31_{(10} = 11111_{(2}$.

- En cuanto a la parte fraccionaria:

$$
\begin{array}{llllll}
0{,}875 \cdot 2 & = & 1{,}75 & = & \mathbf{1} + 0{,}75 & \downarrow \\
0{,}75 \cdot 2 & = & 1{,}5 & = & \mathbf{1} + 0{,}5 & \\
0{,}5 \cdot 2 & = & 1 & = & \mathbf{1} + 0 &
\end{array}
$$

Así pues, $0{,}875_{(10} = 0{,}111_{(2}$

- Finalmente, se juntan todas las partes y obtenemos $+31{,}875_{(10} = \mathbf{0}11111{,}111_{(SM2}$.

Por el $-16{,}21875_{(10}$:

- El bit de signo es 1, porque el número es negativo.
- La parte entera es $16_{(10}$:

$$
\begin{array}{llllll}
16 & = & 8 \cdot 2 & + & \mathbf{0} & \uparrow \\
8 & = & 4 \cdot 2 & + & \mathbf{0} & \\
4 & = & 2 \cdot 2 & + & \mathbf{0} & \\
2 & = & 1 \cdot 2 & + & \mathbf{0} & \\
1 & = & 0 \cdot 2 & + & \mathbf{1} &
\end{array}
$$

Es decir $16_{(10} = 10000_{(2}$.

- En cuanto a la parte fraccionaria:

$$
\begin{array}{llllll}
0{,}21875 \cdot 2 & = & 0{,}4375 & = & \mathbf{0} + 0{,}4375 & \downarrow \\
0{,}4375 \cdot 2 & = & 0{,}875 & = & \mathbf{0} + 0{,}875 & \\
0{,}875 \cdot 2 & = & 1{,}75 & = & \mathbf{1} + 0{,}75 & \\
0{,}75 \cdot 2 & = & 1{,}5 & = & \mathbf{1} + 0{,}5 & \\
0{,}5 \cdot 2 & = & 1 & = & \mathbf{1} + 0 &
\end{array}
$$

Así pues, $0{,}21875_{(10} = 0{,}00111_{(2}$

- Finalmente, se juntan todas las partes y obtenemos $-16{,}21875_{(10} = \mathbf{1}10000{,}00111_{(SM2}$.

Para representar exactamente **los dos** números hace falta un formato que tenga:

- Un bit de signo.

- 5 bits para la parte entera (con 5 bits se pueden representar exactamente tanto $16_{(10}$ como $31_{(10)}$).
- 5 bits para la parte fraccionaria (con 5 bits se pueden representar exactamente tanto $0{,}875_{(10}$ como $0{,}21875_{(10)}$).

Tenemos, pues, un formato de signo y magnitud con $n = 11$ (bit de signo, 5 bits de parte entera y 5 bits de parte fraccionaria) y m = 5. Con estos datos, el rango es el siguiente: $[-2^{n-m-1} + 2^{-m}, +2^{n-m-1} - 2^{-m}] = [-2^{11-5-1} + 2^{-5}, +2^{11-5-1} - 2^{-5}] = [-2^5 + 2^{-5}, +2^5 - 2^{-5}] =$ **$=[-31{,}96875_{(10}, +31{,}96875_{(10}]$**. La precisión es $2^{-m} = 2^{-5} =$ **$0{,}03125_{(10}$**.

**35.** Calculad la suma y la resta de los pares de números siguientes, asumiendo que están en coma fija en signo y magnitud con 8 bits, donde 3 son fraccionarios. Verificad si el resultado es correcto:

**a)** $00111000_{(2}$ y $10100000_{(2}$

$00111000_{(2} \rightarrow +0111{,}000_{(2} = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 4 + 2 + 1 = +7_{(10}$
$10100000_{(2} \rightarrow -0100{,}000_{(2} = -1 \cdot 2^2 = -4_{(10}$

Para calcular la suma de estos dos números, hace falta, primero, darse cuenta de que se trata de números de diferente signo. Por lo tanto, la suma se obtendrá restando la magnitud más grande ($0111{,}000_{(2)}$) de la más pequeña ($0100{,}000_{(2)}$) y copiando el signo de la magnitud más grande (positivo):

```
     0 1 1 1 , 0 0 0 (2
       0 0 0   0 0        ← acarreo
 –   0 1 0 0 , 0 0 0 (2
 ----------------------
     0 0 1 1 , 0 0 0 (2   ← resultado
```

Por lo tanto, $00111000_{(2} + 10100000_{(2} \rightarrow +0111{,}000_{(2} + (-0100{,}000_{(2)}) = +0011{,}000_{(2} \rightarrow$ **$00011000_{(2}$**. Si convertimos el resultado a base 10, tenemos que $+0011{,}000_{(2} = 1 \cdot 2^1 + 1 \cdot 2^0 = 2 + 1 = 3_{(10}$, cosa que demuestra que el resultado es correcto, puesto que $+7_{(10} + (-4_{(10)}) = 3_{(10}$.

Para calcular la resta habrá que hacer la suma de las magnitudes, porque el número $10100000_{(2}$ es negativo. El resultado de la resta será positivo (porque a un número positivo le restamos uno negativo):

```
     1 0 0 0   0 0        ← acarreo
     0 1 1 1 , 0 0 0 (2
 +   0 1 0 0 , 0 0 0 (2
 ----------------------
     1 0 1 1 , 0 0 0 (2   ← resultado
```

Por lo tanto, $00111000_{(2} - 10100000_{(2} \rightarrow +0111{,}000_{(2} - (-0100{,}000_{(2)}) = +1011{,}000_{(2} \rightarrow$ **$01011000_{(2}$**. Nuevamente, podemos comprobar que el resultado obtenido es correcto si lo convertimos a base 10: $+1011{,}000_{(2} = 2^3 + 2^1 + 2^0 = 8 + 2 + 1 = +11_{(10} = +7_{(10} - (-4_{(10)})$.

**b)** $10111010_{(2}$ y $11101100_{(2}$

$10111010_{(2} \rightarrow -0111{,}010_{(2} = -(2^2 + 2^1 + 2^0 + 2^{-2}) = -(4 + 2 + 1 + 0{,}25) = -7{,}25_{(10}$
$11101100_{(2} \rightarrow -1101{,}100_{(2} = -(2^3 + 2^2 + 2^0 + 2^{-1}) = -(8 + 4 + 1 + 0{,}5) = -13{,}5_{(10}$

En este caso se trata de dos números negativos, dado que el primer dígito, que indica el signo, es un 1 en ambos casos. Para hacer la suma, habrá que hacer la suma de las magnitudes y copiar el bit signo:

```
       1 1 1 0   0 0        ← acarreo
       0 1 1 1 , 0 1 0 (2
 +     1 1 0 1 , 1 0 0 (2
 ------------------------
     1 0 1 0 0 , 1 1 0 (2   ← resultado
```

Notad que se ha producido acarreo en el último bit y, por lo tanto, como estamos sumando sólo las magnitudes, hay desbordamiento. Esto quiere decir que el resultado de la suma no es representable con 8 bits, de los cuales 3 son fraccionarios.

Si dispusiéramos de un bit más para la parte entera (un formato de 9 bits con 3 de ellos fraccionarios) el resultado sí que sería representable: tendríamos $-10100{,}110_{(2} \rightarrow 110100110$. Efectivamente, comprobamos que
$-10100{,}110_{(2} = -(2^4 + 2^2 + 2^{-1} + 2^{-2}) = -(16 + 4 + 0{,}5 + 0{,}25) = -20{,}75_{(10} = -7{,}25_{(10} + (-13{,}5_{(10)})$.

En cuanto a la resta, hay que darse cuenta de que el sustraendo ($-1101{,}100_{(2)}$) es más grande en magnitud que el minuendo ($-0111{,}010_{(2)}$). Así pues, haremos la resta de las magnitudes como $1101{,}100_{(2} - 0111{,}010_{(2}$ y el resultado tiene que ser positivo:

$$
\begin{array}{ccccccccccl}
 & 1 & 1 & 0 & 1 & , & 1 & 0 & 0 & _{(2} & \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & & {\scriptstyle 1} & {\scriptstyle 0} & & & \leftarrow \text{acarreo} \\
- & 0 & 1 & 1 & 1 & , & 0 & 1 & 0 & _{(2} & \\
\hline
 & 0 & 1 & 1 & 0 & , & 0 & 1 & 0 & _{(2} & \leftarrow \text{resultado}
\end{array}
$$

Por lo tanto, $10111{,}010_{(2} - 11101100_{(2} \rightarrow -0111{,}010_{(2} - (-1101100_{(2)}) = +0110{,}010_{(2} \rightarrow$ $\mathbf{0110010}_{(2}$. Nuevamente, podemos comprobar que el resultado obtenido es correcto si lo convertimos a base 10:
$+0110{,}010_{(2} = 2^2 + 2^1 + 2^{-2} = 4 + 2 + 0{,}25 = +6{,}25_{(10} = -7{,}25_{(10} - (-13{,}5_{(10)}) = -7{,}25_{(10} + 13{,}5_{(10}$

**36.** Codificad en BCD el número $125_{(10}$

En primer lugar codificamos en binario con 4 bits cada uno de los dígitos del número:

$1_{(10} = 0001_{(2}$
$2_{(10} = 0010_{(2}$
$5_{(10} = 0101_{(2}$

Y, finalmente, formamos una única tira con todos los bits $\mathbf{000100100101}_{(2}$.

**37.** Codificad en BCD el número $637_{(10}$

Como en el apartado anterior, primeramente codificamos en binario con 4 bits cada uno de los dígitos del número:

$6_{(10} = 0110_{(2}$
$3_{(10} = 0011_{(2}$
$7_{(10} = 0111_{(2}$

Y, finalmente, formamos una única tira con todos los bits $\mathbf{011000110111}_{(2}$.

**38.** Indicad qué número codifica la representación BCD siguiente 00010011100.

Separamos la tira de bits en grupos de 4 y comprobamos qué número decimal representan. Tenemos $000100111000_{(2} \rightarrow 0001 \mid 0011 \mid 1000$, por lo tanto:

$0001_{(2} = 1_{(10}$
$0011_{(2} = 3_{(10}$
$1000_{(2} = 8_{(10}$

Y, finalmente, construimos el número decimal juntando los dígitos decimales: $\mathbf{138}_{(10}$.

**39.** Codificad el número $427_{(10}$ en BCD y en binario. Comparad el número de bits necesario en los dos casos.

Para codificar $427_{(10}$ en BCD, primeramente codificamos en binario con 4 bits cada uno de los dígitos del número:

$4_{(10} = 0100_{(2}$
$2_{(10} = 0010_{(2}$
$7_{(10} = 0111_{(2}$

Y, finalmente, formamos una única tira con todos los bits $\mathbf{010000100111}_{(2}$.

Para codificar el mismo número en binario, hay que aplicar el método de las divisiones sucesivas por la base de llegada (2):

$$\begin{array}{rcrcccl} 427 & = & 213 & \cdot & 2 & + & \mathbf{1} \\ 213 & = & 106 & \cdot & 2 & + & \mathbf{1} \\ 106 & = & 53 & \cdot & 2 & + & \mathbf{0} \\ 53 & = & 26 & \cdot & 2 & + & \mathbf{1} \\ 26 & = & 13 & \cdot & 2 & + & \mathbf{0} \\ 13 & = & 6 & \cdot & 2 & + & \mathbf{1} \\ 6 & = & 3 & \cdot & 2 & + & \mathbf{0} \\ 3 & = & 1 & \cdot & 2 & + & \mathbf{1} \\ 1 & = & 0 & \cdot & 2 & + & \mathbf{1} \end{array} \uparrow$$

Por lo tanto, $427_{(10} = \mathbf{110101011}_{(2}$.

En el caso de la codificación BCD, hacen falta **12 bits** para representar el número $427_{(10}$, en cambio, la codificación binaria sólo necesita **9**.

**40.** Encontrad el valor decimal que codifican las cadenas de bits siguientes, interpretando que se trata de números en un formato de coma flotante de 8 bits con mantisa normalizada de la forma 1,*X* y con bit implícito:

**a)** 11110010, donde la mantisa es de 4 bits.

En primer lugar hay que identificar los diferentes elementos que componen este formato:

| Signo (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 | 111 | 0010 |

- Bit de signo $s = 1$, por lo tanto, se trata de un número negativo.
- Exponente: como la mantisa es de 4 bits, quedan 3 bits ($111_{(2)}$) para el exponente ($q = 3$). Por lo tanto, el exceso será $M = 2^{q-1} = 2^2 = 4_{(10}$. Entonces, el exponente vale $e = 111_{(2} - 4_{(10} = (2^2 + 2^1 + 2^0) - 4 = (4 + 2 + 1) - 4 = 7 - 4 = 3_{(10}$.
- Finalmente, los cuatro bits que quedan forman la mantisa que, como está normalizada en la forma 1,*X* con bit implícito toma el valor $R = 1{,}0010_{(2}$.

Ahora ya podemos calcular el número representado en base 10 aplicando la fórmula $\pm R \cdot 2^e$ y el TFN:

$$-1{,}0010_{(2} \cdot 2^3 = -(2^0 + 2^{-3}) \cdot 2^3 = -(2^3 + 2^0) = -(8 + 1) = \mathbf{-9_{(10}}.$$

**b)** 01010011, donde la mantisa es de 3 bits.

En este caso, procedemos de manera totalmente análoga al apartado anterior:

| Signo (*S*) | Exponente (*e*) | Mantissa |
|---|---|---|
| 0 | 1010 | 011 |

- Bit de signo $s = 0$, por lo tanto, se trata de un número positivo.
- Exponente: como la mantisa es de 3 bits, quedan 4 bits ($1010_{(2)}$) para el exponente ($q = 4$). Por lo tanto, el exceso será $M = 2^{q-1} = 2^3 = 8_{(10}$. Entonces, el exponente vale $e = 1010_{(2} - 8_{(10} = (2^3 + 2^1) - 8 = (8 + 2) - 8 = 10 - 8 = 2_{(10}$.
- Finalmente, los cuatro bits que quedan forman la mantisa que, como está normalizada en la forma 1,*X* con bit implícito toma el valor $R = 1{,}011_{(2}$.

Ahora ya podemos calcular el número representado en base 10 aplicando la fórmula $\pm R \cdot 2^e$ y el TFN:

$$+1{,}011_{(2} \cdot 2^2 = +(2^0 + 2^{-2} + 2^{-3}) \cdot 2^2 = +(2^2 + 2^0 + 2^{-1}) = +(4 + 1 + 0{,}5) = \mathbf{+5{,}5_{(10}}.$$

**41.** Haced las codificaciones siguientes:

**a)** El número $-1{,}335_{(10}$ en coma flotante de 8 bits, mantisa de 3 bits normalizada de la forma 1,*X* y con bit implícito empleando una aproximación por truncamiento.

El formato especificado tiene la forma dada por la tabla siguiente:

| Signo (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 bit | 4 bits | 3 bits |

En primer lugar, codificaremos la magnitud $1{,}335_{(10}$ en base 2:

- La parte entera es $1_{(10}$:

$$1 = 0 \cdot 2 + \mathbf{1}$$

Es decir $1_{(10} = 1_{(2}$.

- En cuanto a la parte fraccionaria:

$$\begin{array}{lcccccl} 0{,}335 & \cdot & 2 & = & 0{,}67 & = & \mathbf{0} + 0{,}67 \\ 0{,}67 & \cdot & 2 & = & 1{,}34 & = & \mathbf{1} + 0{,}34 \\ 0{,}34 & \cdot & 2 & = & 0{,}68 & = & \mathbf{0} + 0{,}68 \\ 0{,}68 & \cdot & 2 & = & 1{,}36 & = & \mathbf{1} + 0{,}36 \\ 0{,}36 & \cdot & 2 & = & 0{,}72 & = & \mathbf{0} + 0{,}72 \\ 0{,}72 & \cdot & 2 & = & 1{,}44 & = & \mathbf{1} + 0{,}44 \end{array} \downarrow$$

Así pues, $0{,}355_{(10} = 0{,}010101\ldots_{(2}$ y paramos porque ya tenemos bastantes más bits de los que permite representar este formato.

- Ahora, se juntan las dos partes de la magnitud y obtenemos $1{,}335_{(10} \approx 1{,}010101_{(2}$.

- A continuación normalizamos la expresión obtenida (añadiendo la información del signo) la forma $\pm R \cdot 2^e$: $+1{,}335_{(10} \approx +1{,}010101_{(2} \cdot 2^0$.

- Finalmente, obtenemos los diferentes elementos que definen la representación:

– **Signo**: se trata de un número negativo, por lo tanto, el bit de signo será un **1**.

– **Exponente**: El exponente es $0_{(10}$ que hay que codificar en exceso a $M = 2^{q-1}$. Como disponemos de 4 bits, el exceso es $M = 2^3 = 8$. Por lo tanto hay que representar $0_{(10} + 8_{(10} = 8_{(10}$ con 4 bits:

$$\begin{array}{ccccc} 8 & = & 4 \cdot 2 & + & \mathbf{0} \\ 4 & = & 2 \cdot 2 & + & \mathbf{0} \\ 2 & = & 1 \cdot 2 & + & \mathbf{0} \\ 1 & = & 0 \cdot 2 & + & \mathbf{1} \end{array} \uparrow$$

Es decir $e = 8_{(10} = \mathbf{1000_{(2}}$ (en exceso a 8).

– **Mantisa**: $R = 1{,}010101_{(2}$. Como hay bit implícito ($1{,}X$), sólo hay que representar la parte de la derecha de la coma y disponemos de 3 bits (con truncamiento). Así pues la mantisa serán los tres bits a partir de la coma: **010**.

– Ahora ya podemos juntar todas las partes:

| Signo (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| **1** | **1000** | **010** |

Es decir, la representación es **11000010**.

**b)** Repetid el apartado anterior, pero con una aproximación por redondeo.

La única diferencia con el apartado anterior es el cálculo de la mantisa, porque ahora hay que redondear en vez de truncar. Tal y como se indica más arriba, tenemos $R = 1{,}010101_{(2}$, para redondearlo con tres bits, hay que sumar la mitad de la precisión y truncar el resultado:

$$1{,}010101_{(2} + 0{,}0001_{(2} = 1{,}011001_{(2} \rightarrow \text{truncamos 3 bits} \rightarrow 1{,}011_{(2}$$

Por lo tanto, la representación será la siguiente:

| Signo (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| **1** | **1000** | **011** |

Es decir, la secuencia de bits de la representación es **11000011**.

**c)** El número $10{,}0327_{(10}$ en coma flotante de 9 bits, mantisa de 3 bits normalizada de la forma $1{,}X$ y con bit implícito empleando una aproximación por truncamiento.

El formato especificado tiene la forma dada por la tabla siguiente:

| Signe (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 bit | 5 bits | 3 bits |

En primer lugar, codificaremos la magnitud $10{,}0327_{(10}$ en base 2:

- La parte entera es $10_{(10}$:

$$\begin{array}{rcrcl} 10 & = & 5 \cdot 2 & + & \mathbf{0} \\ 5 & = & 2 \cdot 2 & + & \mathbf{1} \\ 2 & = & 1 \cdot 2 & + & \mathbf{0} \\ 1 & = & 0 \cdot 2 & + & \mathbf{1} \end{array} \uparrow$$

Es decir $10_{(10} = 1010_{(2}$.

- En cuanto a la parte fraccionaria:

$$\begin{array}{lcccccl} 0{,}0327 & \cdot & 2 & = & 0{,}0654 & = & \mathbf{0} + 0{,}0654 \\ 0{,}0654 & \cdot & 2 & = & 0{,}1308 & = & \mathbf{0} + 0{,}1308 \\ 0{,}1308 & \cdot & 2 & = & 0{,}2616 & = & \mathbf{0} + 0{,}2616 \\ 0{,}2616 & \cdot & 2 & = & 0{,}5232 & = & \mathbf{0} + 0{,}5232 \\ 0{,}5232 & \cdot & 2 & = & 1{,}0464 & = & \mathbf{1} + 0{,}0464 \\ 0{,}0464 & \cdot & 2 & = & 0{,}0928 & = & \mathbf{0} + 0{,}0928 \end{array} \downarrow$$

Así pues, $0{,}0327_{(10} = 0{,}000010..._{(2}$ y paramos porque ya tenemos bastantes más bits de los que permite representar este formato.

- Ahora, se juntan las dos partes de la magnitud y obtenemos $10{,}0327_{(10} \approx 1010{,}000010_{(2}$.

- A continuación normalizamos la expresión obtenida (añadiendo la información del signo) en la forma $\pm R \cdot 2^e$: $+10{,}0327_{(10} \approx +1{,}010000010_{(2} \cdot 2^3$.

- Finalmente, obtenemos los diferentes elementos que definen la representación:

– **Signo**: se trata de un número positivo, por lo tanto, el bit de signo será un **0**.

– **Exponente**: El exponente es $3_{(10}$ que hay que codificar en exceso a $M = 2^{q-1}$. Como disponemos de 5 bits el exceso es $M = 2^4 = 16$. Por lo tanto hay que representar $3_{(10} + 16_{(10} = 19_{(10}$ con 4 bits:

$$\begin{array}{rcrcl} 19 & = & 9 \cdot 2 & + & \mathbf{1} \\ 9 & = & 4 \cdot 2 & + & \mathbf{1} \\ 4 & = & 2 \cdot 2 & + & \mathbf{0} \\ 2 & = & 1 \cdot 2 & + & \mathbf{0} \\ 1 & = & 0 \cdot 2 & + & \mathbf{1} \end{array} \uparrow$$

Es decir $e = 19_{(10} = \mathbf{10011}_{(2}$ (en exceso a 16).

– **Mantisa**: $R = 1{,}010000010_{(2}$. Como hay bit implícito (1,*X*), sólo hay que representar la parte de la derecha de la coma y disponemos de 3 bits (con truncamiento). Así pues la mantisa serán los tres bits a partir de la coma: **010**.

– Ahora ya podemos juntar todas las partes:

| Signo (*S*) | Exponente (*e*) | Mantisa |
|---|---|---|
| **0** | **10011** | **010** |

Es decir, la representación es **0$\underline{10011}$010**.

**42.** Determinad si el número $\mathbf{2{,}89_{(10} \cdot 10^{10}}$ es representable en un formato de coma flotante de 16 bits, con mantisa normalizada de la forma 1,*X*, bit implícito y 5 bits para el exponente.

El formato especificado tiene la forma dada por la tabla siguiente:

| Signo (*s*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 bit | 5 bits | 10 bits |

El número positivo de magnitud mayor que podemos representar en este formato tiene un cero en el bit de signo y un 1 en el resto de bits: 0<u>11111</u>1111111111.

Buscamos el número decimal que representa:
- Bit de signo $s = 0$, por lo tanto, se trata de un número positivo.
- Exponente: $11111_{(2}$ , es decir $q = 5$. Por tanto, el exceso será $M = 2^{q-1} = 2^4 = 16_{(10}$. Entonces, el exponente vale $e = 11111_{(2} - 16_{(10} = 31 - 16 = 15_{(10}$.
- Finalmente, los bits que quedan forman la mantisa, que, como está normalizada en la forma 1,*X* con bit implícito, toma el valor $R = 1{,}1111111111_{(2}$.

Ahora ya podemos calcular el número representado en base 10 aplicando la fórmula $\pm R{\cdot}2^e$ y el TFN:

$+1{,}1111111111_{(2}{\cdot}2^{15} = +11111111111_{(2}{\cdot}2^5 = +\,2047{\cdot}\,2^5 = +65504_{(10}$.

Como el número $2{,}89_{(10}{\cdot}10^{10}$ es mayor que el $65504_{(10}$, no se puede representar en este formato.

**43.** Determinad si el número $\mathbf{-1256_{(10}{\cdot}10^{-2}}$ es representable en un formato de coma flotante de 10 bits, con mantisa normalizada de la forma 1,*X*, bit implícito y 6 bits para el exponente.

El formato especificado tiene la forma dada por la tabla siguiente:

| Signo (*s*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 bit | 6 bits | 3 bits |

El número negativo de magnitud mayor que podemos representar en este formato tiene todos los bits a 1: 1<u>111111</u>111.

Buscamos el número decimal que representa:
- Bit de signo $s = 1$, por lo tanto, se trata de un número negativo.
- Exponente: $111111_{(2}$ , es decir $q = 6$. Por tanto, el exceso será $M = 2^{q-1} = 2^5 = 32_{(10}$. Entonces, el exponente vale $e = 111111_{(2} - 32_{(10} = 63 - 32 = 31_{(10}$.
- Finalmente, los bits que quedan forman la mantisa, que, como está normalizada en la forma 1,*X* con bit implícito, toma el valor $R = 1{,}111_{(2}$.

Ahora ya podemos calcular el número representado en base 10 aplicando la fórmula $\pm R{\cdot}2^e$ y el TFN:
$-1{,}111_{(2}{\cdot}2^{31} = -1111_{(2}{\cdot}2^{28} = -15{\cdot}\,2^{28}{}_{(10}$.

Como la magnitud del número $-1256_{(10}{\cdot}10^{-2}$ (o lo que es lo mismo, del $-12{,}56_{(10}$) es más pequeña que $-15{\cdot}\,2^{28}{}_{(10}$, se puede representar en este formato.

## Ejercicios de autoevaluación

**1.** Podemos obtener la codificación en Ca2 y 8 bits del número $-10_{(10}$ de las siguientes formas :

- $2^8 - 10 = 256_{(10} - 10_{(10} = 246_{(10} = \mathbf{11110110}_{(Ca2}$ , o bien,
- $2^8 - 10 = 100000000_{(2} - 1010_{(2} = \mathbf{11110110}_{(Ca2}$ , o bien,
- $+10_{(10} = +1010_{(2}$ → Representación de la magnitud positiva → $00001010_{(Ca2}$ →
  → Cambio de signo → $\mathbf{11110110_{(Ca2}}$

Para codificar el número en signo y magnitud, hemos de añadir a la representación de la magnitud en base 2 la información del signo. Como es un número negativo, el bit de signo será 1. Por lo tanto, la codificación será $\mathbf{10001010_{(SM2}}$

**2.**
**a)** Si se trata de un número codificado en complemento a 2.
En este caso se trata de un número positivo porque el bit del extremo izquierdo es 0. Por lo tanto, el resto de los bits codifican la magnitud como en el caso de signo y magnitud. Si aplicamos el TFN a la magnitud, obtenemos:

$0100100_{(2} = 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 36_{(10}$

En este caso codifica el número decimal +36.

**b)** Si se trata de un número codificado en signo y magnitud.
Como se trata de un número positivo, y la codificación en signo y magnitud de un número positivo coincide con la representación en Ca2, la cadena codifica el mismo número, el $+36_{(10}$.

**3.**

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | | | | | | | | | |
| | | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 (2 |
| + | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 (2 |
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 (2 |

Lo que podemos observar es que necesitamos un bit más para poder representar el resultado de la operación.

**4.** Como se trata de un número positivo, el bit de signo es **0**. Con respecto a la magnitud $12{,}346_{(10}$, primero pasamos a binario la parte entera y, posteriormente, la parte fraccionaria. Para la parte entera utilizamos el algoritmo de divisiones sucesivas por la base de llegada (2):

$$\begin{aligned} 12 &= 6 \cdot 2 + \mathbf{0} \\ 6 &= 3 \cdot 2 + \mathbf{0} \\ 3 &= 1 \cdot 2 + \mathbf{1} \\ 1 &= 0 \cdot 2 + \mathbf{1} \end{aligned} \uparrow$$

Así pues, $12_{(10} = 1100_{(2}$. Con respecto a la parte fraccionaria, hacemos multiplicaciones sucesivas por la base de llegada (2):

$$\begin{aligned} 0{,}346 \cdot 2 &= 0{,}692 = \mathbf{0} + 0{,}692 \\ 0{,}692 \cdot 2 &= 1{,}384 = \mathbf{1} + 0{,}384 \\ 0{,}384 \cdot 2 &= 0{,}768 = \mathbf{0} + 0{,}768 \\ 0{,}768 \cdot 2 &= 1{,}536 = \mathbf{1} + 0{,}536 \\ 0{,}536 \cdot 2 &= 1{,}072 = \mathbf{1} + 0{,}072 \end{aligned} \downarrow$$

Como el formato especificado sólo tiene 3 bits para la parte fraccionaria, ya tenemos más bits de los necesarios y podemos detener el proceso aquí. Por lo tanto, $0{,}346_{(10} = 0{,}01011..._{(2}$. Para aproximar el valor con 3 bits por truncamiento eliminamos todos los bits a partir del tercero: $0{,}346_{(10} = 0{,}010_{(2}$

A continuación, unimos las partes entera y fraccionaria y obtenemos: $12{,}346_{(10} \approx 1100{,}010_{(2}$. Finalmente, para obtener la representación en el formato indicado, hay que añadir el bit de signo, de manera que la tira de bits que representa este número en el formato dado es: $01100{,}010_{(2}$. Hay que recordar que la tira de bits que se almacenaría en un computador no contiene la coma ni la especificación de la base: **01100010**.

**5.** Podemos obtener la representación en Ca2 y 8 bits del $-45_{(10}$ de cualquiera de las siguientes formas:

- $2^8 - 45 = 256_{(10} - 45_{(10} = 211_{(10} = \mathbf{11010011}_{(Ca2}$ , o bien,
- $2^8 - 45 = 100000000_{(2} - 101101_{(2} = \mathbf{11010011}_{(Ca2}$ , o bien,
- $+45_{(10} = +101101_{(2}$ → Representación de la magnitud positiva → $00101101_{(Ca2}$ → → cambio de signo → $\mathbf{11010011}_{(Ca2}$

**6.** En primer lugar, obtendremos la codificación binaria del número $-35{,}25_{(10}$:

- El bit de signo es 1 porque el número es negativo.
- La parte entera es $35_{(10}$:

$$\begin{aligned} 35 &= 17 \cdot 2 + 1 \\ 17 &= 8 \cdot 2 + 1 \\ 8 &= 4 \cdot 2 + 0 \\ 4 &= 2 \cdot 2 + 0 \\ 2 &= 1 \cdot 2 + 0 \\ 1 &= 0 \cdot 2 + 1 \end{aligned} \uparrow$$

Es decir, $35_{(10} = 100011_{(2}$.

- Con respecto a la parte fraccionaria:

$$\begin{aligned} 0{,}25 \cdot 2 &= 0{,}5 = \mathbf{0} + 0{,}5 \\ 0{,}5 \cdot 2 &= 1 = \mathbf{1} + 0 \end{aligned} \downarrow$$

Así pues, $0{,}25_{(10} = 0{,}01_{(2}$

- Finalmente, se unen todas las partes y obtenemos $-35{,}25_{(10} = 1100011{,}01_{(SM2}$.

Con el fin de representar exactamente este número, es necesario un formato que tenga un total de 9 bits:
- Un bit de signo
- 6 bits para la parte entera
- 2 bits para la parte fraccionaria

**7.** La equivalencia binaria de estos números decimales es la siguiente:
$+12{,}25_{(10} = +1100{,}01_{(2}$
$+32{,}5_{(10} = +100000{,}1_{(2}$

En signo y magnitud y 9 bits donde 2 son fraccionarios, la codificación es la siguiente:
$+1100{,}01_{(2} = 0001100{,}01_{(SM2}$
$+100000{,}1_{(2} = 0100000{,}10_{(SM2}$

Para hacer la suma examinamos los bits de signo. Se trata de dos números positivos, por lo tanto, se deben sumar las magnitudes y añadir al resultado un bit de signo positivo:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 0 | 0, | 0 | 1 | (2 |
| + | 1 | 0 | 0 | 0 | 0 | 0, | 1 | 0 | (2 |
| | 1 | 0 | 1 | 1 | 0 | 0, | 1 | 1 | (2 |

Al hacer la suma no se produce desbordamiento porque no hay acarreo en la última etapa, que es lo que determina si hay desbordamiento en este formato. Al resultado obtenido se debe añadir el bit de signo para tener la codificación en signo y magnitud correcta: $\mathbf{0101100{,}11_{(SM2}}$

**8.** Para codificar en BCD tan sólo hemos de codificar cada dígito decimal con 4 bits:
$1 = 0001_{(2}$
$7 = 0111_{(2}$
$8 = 1000_{(2}$

Finalmente, unimos los códigos BCD para obtener la cadena final: **000101111000**

**9.** El formato especificado tiene la forma dada por la tabla siguiente:

| Signo (*s*) | Exponente (*e*) | Mantisa |
|---|---|---|
| 1 bit | 4 bits | 3 bits |

- En primer lugar, hemos de codificar la magnitud $12{,}346_{(10}$ en base 2. En el ejercicio de autoevaluación 4 hemos obtenido que $12{,}346_{(10} \approx 1100{,}010_{(2}$
- Seguidamente normalizamos la expresión obtenida en la forma $\pm R \cdot 2^e$:
$+12{,}346_{(10} \approx +1{,}100010_{(2} \cdot 2^3$.

- Finalmente, obtenemos los diferentes elementos que definen la representación:
  - **Signo**: se trata de un número positivo, por tanto, el bit de signo será **0**.
  - **Exponente**: el exponente es $3_{(10}$, que hay que codificar en exceso en $M = 2^{q-1}$. Como disponemos de 4 bits, el exceso es $M = 2^3 = 8$. Por lo tanto, hay que representar $3_{(10} + 8_{(10} = 11_{(10}$ con 4 bits:

$$\begin{aligned} 11 &= 5 \cdot 2 + 1 \\ 5 &= 2 \cdot 2 + 1 \\ 2 &= 1 \cdot 2 + 0 \\ 1 &= 0 \cdot 2 + 1 \end{aligned} \uparrow$$

Es decir $e = 11_{(10} = \mathbf{1011_{(2}}$ (en exceso a 8).
  - **Mantisa**: $R = 1{,}100010_{(2}$. Como hay bit implícito $(1,X)$, sólo hay que representar la parte de la derecha de la coma y disponemos de 3 bits (con truncamiento). Así pues, la mantisa serán los tres bits a partir de la coma: **100**.
  - Ahora ya podemos juntar todas las partes:

| Signo (*s*) | Exponente (*e*) | Mantisa |
|---|---|---|
| **0** | **1011** | **100** |

Es decir, la representación es **0<u>1011</u>100**.

**10.** En primer lugar, desempaquetamos la cadena de bits. Cada dígito hexadecimal corresponde a 4 bits:

- 3 = 0011
- 7 = 0111
- 8 = 1000

Por lo tanto, $378_{(16}$ empaqueta la cadena de bits 001101111000. Esta cadena codifica un número en coma flotante con 4 bits de mantisa: 001101111000. Buscamos el número decimal que representa:

- Bit de signo $s = 0$, por lo tanto, se trata de un número positivo.
- Exponente: $0110111_{(2}$, es decir $q = 7$. Por lo tanto, el exceso será $M = 2^{q-1} = 2^6 = 64_{(10}$. Entonces, el exponente vale $e = 0110111_{(2} - 64_{(10} = 55 - 64 = -9_{(10}$.
- Finalmente, los bits que quedan forman la mantisa, que, como está normalizada en la forma $1,X$ con bit implícito, toma el valor $R = 1{,}1000_{(2}$.

Ahora ya podemos calcular el número representado en base 10 aplicando la fórmula $\pm R \cdot 2^e$ y el TFN:

$$-1{,}1000_{(2} \cdot 2^{-9} = -11_{(2} \cdot 2^{-10} = -3_{(10} \cdot 2^{-10}{}_{(10} = -\frac{3}{1024}$$

## Glosario

**acarreo** *m* Bit que indica que se ha superado el valor *b* de la base de numeración al sumar dos dígitos. La suma de dos valores numéricos se lleva a cabo dígito a dígito, progresando de derecha a izquierda. Un suma y sigue al sumar dos dígitos indica que hay que añadir una unidad al hacer la suma de los dos dígitos siguientes.
*en* carry

**base** *f* Número de dígitos diferentes en un sistema de numeración.

***binary coded decimal*** **(BCD)** Véase **decimal codificado en binario**.

**binario** *m* Sistema de numeración en base 2.

**bit** *m* Abreviación de *BInary DigiT* ('dígito binario'). Corresponde a la unidad de información más pequeña posible. Se define como la cantidad de información asociada a la respuesta de una pregunta formulada de una manera no ambigua, donde sólo son posibles dos alternativas de respuesta, y que, además, tienen la misma probabilidad de ser escogidas.

**bit menos significativo** *m* Bit menos significativo (más a la derecha) de una representación numérica posicional.
**sigla: LSB**.

**bit más significativo** *loc* Bit más significativo (más a la izquierda) de una representación numérica posicional.
**sigla: MSB**.

***borrow*** *m* Suma y sigue empleado en la operación de resta.

**byte** Véase **octeto**.

**cadena** *f* Conjunto de elementos de un mismo tipo dispuestos uno a continuación del otro.

**carry** Véase **acarreo**.

**coma fija** *f* Sistema de representación numérica que emplea un número fijo de dígitos enteros y fraccionarios.

**coma flotante** *f* Sistemas de representación numérica que codifican un número variable de dígitos enteros y fraccionarios. La cantidad de dígitos enteros y fraccionarios depende del formato y del valor numérico que se representa.

**decimal** *m* Sistema de numeración en base 10.

**decimal codificado en binario** *m* Designa la codificación de los dígitos decimales (0,1,2...,9) sobre un conjunto de 4 dígitos binarios.
**sigla: BCD**

**desbordamiento** *m* Indicación de que el resultado de una operación está fuera del rango de representación disponible.
*en* overflow

**dígito** *m* Elemento de información que puede coger un número finito de valores diferentes. La cantidad de valores diferentes es dada por la base de numeración.

**digital** *m* Sistema que trabaja con datos discretos.

**empaquetamiento** *m* Transformación que permite representar la información binaria de forma más compacta.

**formato** *m* Descripción estructural de una secuencia de datos, donde se especifica el tipo, la longitud y la disposición de cada elemento.

**fraccionario** *m* Menor que la unidad.

**hexadecimal** *m* Sistema de numeración en base 16.

**LSB** Véase **bit menos significativo**

**palabra** *f* Unidad de información procesada por un computador de un solo golpe (en paralelo). *en* word

**MSB** Véase **bit más significativo**.

**octal** *m* Sistema de numeración en base 8.

**octeto** *m* Cadenas de 8 bits.

***overflow*** Véase desbordamiento.

**precisión** *f* Diferencia entre dos valores consecutivos de una representación.

**raíz** *f* Base de un sistema de numeración.

**rango** *m* Conjunto de valores que indican los márgenes entre los cuales se encuentran los valores posibles de un formato de representación.

**representación** Véase formato.

**word** Véase **palabra**.

## Bibliografía

**De Miguel, P.** (1996). *Fundamentos de los computadores* (5.ª ed., cap. 2). Madrid: Paraninfo.

**Patterson, D.; Henessy, J. L.** (1995). *Organización y diseño de computadores: la interfaz hardware/software*. Madrid: McGraw-Hill.

**Stallings, W.** (1996). *Organización y arquitectura de computadores: diseño para optimizar prestaciones*. Madrid: Prentice Hall.

# Los circuitos lógicos combinacionales

Montse Peiron Guàrdia
Fermín Sánchez Carracedo

PID_00163599

Universitat Oberta de Catalunya

www.uoc.edu

# Índice

**Introducción** .......... 5

**Objetivos** .......... 6

**1. Fundamentos de la electrónica digital** .......... 7
1.1. Circuitos, señales y funciones lógicas .......... 7
1.2. Álgebra de Boole .......... 10
1.3. Representación de funciones lógicas .......... 13
1.3.1. Expresiones algebraicas .......... 13
1.3.2. Tablas de verdad .......... 15
1.3.3. Correspondencia entre expresiones algebraicas y tablas de verdad .......... 17
1.3.4. Expresiones en suma de mintérminos .......... 18
1.4. Otras funciones comunes .......... 20
1.5. Funciones especificadas de manera incompleta .......... 21

**2. Implementación de circuitos lógicos combinacionales** .......... 24
2.1. Puertas lógicas. Síntesis y análisis .......... 24
2.2. Diseño de circuitos a dos niveles .......... 28
2.2.1. Retardos. Cronogramas. Niveles de puertas .......... 28
2.2.2. Síntesis a dos niveles .......... 30
2.3. Minimización de funciones .......... 31
2.3.1. Simplificación de expresiones .......... 31
2.3.2. Síntesis mínima a dos niveles. Método de Karnaugh .......... 34
2.3.3. Minimización de funciones especificadas incompletamente .......... 39

**3. Bloques combinacionales** .......... 41
3.1. Multiplexor. Multiplexor de buses. Demultiplexor .......... 41
3.2. Codificadores y descodificadores .......... 46
3.3. Desplazadores lógicos y aritméticos .......... 50
3.4. Bloques AND, OR y NOT .......... 51
3.5. Memoria ROM .......... 53
3.6. Comparador .......... 55
3.7. Sumador .......... 56
3.8. Unidad aritmética y lógica (UAL) .......... 58

**Resumen** .......... 61

**Ejercicios de autoevaluación** .......... 63

**Solucionario** .......... 66

**Bibliografía** .......... 107

# Introducción

Un computador es una máquina construida a partir de dispositivos electrónicos básicos, adecuadamente interconectados. Podemos decir que estos dispositivos constituyen las “piezas” o “ladrillos” con los que se construye un computador.

En este módulo conoceremos a fondo los dispositivos electrónicos básicos. Los dispositivos electrónicos más elementales son las **puertas lógicas** y los **bloques** lógicos, que forman los **circuitos lógicos**. Estos últimos se pueden ver como un conjunto de dispositivos que manipulan de una manera determinada las señales electrónicas que les llegan (las **señales de entrada**), y generan como resultado otro conjunto de señales (las **señales de salida**).

Existen dos grandes tipos de circuitos lógicos:

**1)** Los **circuitos combinacionales**, que se caracterizan porque el valor de las señales de salida en un momento determinado depende del valor de las señales de entrada en ese mismo momento.

**2)** Los **circuitos secuenciales**, en los que el valor de las señales de salida en un momento determinado depende de los valores que han llegado por las señales de entrada desde la puesta en funcionamiento del circuito. Por tanto, tienen capacidad de memoria.

En este módulo se estudian los circuitos lógicos combinacionales. Los circuitos secuenciales se estudiarán en el siguiente módulo “Los circuitos lógicos secuenciales”.

## Objetivos

El objetivo fundamental de este módulo es conocer a fondo los circuitos lógicos combinacionales, es decir, saber cómo están formados y ser capaces de utilizarlos con agilidad, hasta el punto de familiarizarnos totalmente con ellos.

Para llegar a este punto se tendrán que haber conseguido los siguientes objetivos:

1. Entender el álgebra de Boole y las diferentes maneras de expresar funciones lógicas.

2. Conocer las diferentes puertas lógicas, ver cómo se pueden utilizar para sintetizar funciones lógicas y ser capaces de hacerlo. Entender por qué se desea minimizar el número de puertas y de niveles de puertas de los circuitos y saberlo hacer.

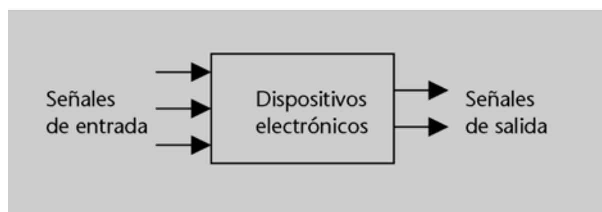
3. Conocer la funcionalidad de un conjunto de bloques combinacionales básicos y ser capaces de utilizarlos en el diseño de circuitos.

En definitiva, después del estudio de este módulo debemos ser capaces de construir fácilmente un circuito cualquiera usando los diferentes dispositivos que se habrán conocido y entender la funcionalidad de cualquier circuito dado.

# 1. Fundamentos de la electrónica digital

## 1.1. Circuitos, señales y funciones lógicas

Entendemos por circuito un sistema formado por un cierto número de **señales de entrada** (cada señal corresponde a un cable), un conjunto de **dispositivos electrónicos** que hacen operaciones sobre las señales de entrada (las manipulan electrónicamente) y que generan un determinado número de **señales de salida**. Las señales de salida, pues, se pueden considerar como funciones de las de entrada y se puede decir que los dispositivos electrónicos *computan* estas funciones.




En los circuitos, los cables se pueden encontrar en dos valores de tensión (voltaje): tensión alta o tensión baja. Estos dos valores se identifican normalmente con los símbolos 1 y 0, respectivamente, de manera que se dice que una señal *vale 0* (cuando en el cable correspondiente hay tensión baja) o *vale 1* (cuando en el cable correspondiente hay tensión alta, también llamada *tensión de alimentación*); cuando una señal vale 1, se dice que está activa. Las señales que pueden tomar los valores 0 ó 1 se denominan **señales lógicas** o **binarias**. Un circuito lógico es aquél en el que las señales de entrada y de salida son lógicas. Las funciones que computa un circuito lógico son **funciones lógicas**.

> La tensión alta o tensión de alimentación base puede ser de 3,3, 5 o 12 voltios, pero actualmente los circuitos tienen dispositivos para modificarla según los requerimientos de cada componente y de cada momento; por ejemplo, cuando el procesador está en una fase de actividad alta el voltaje aumenta.
> La tensión baja siempre es de 0 voltios.

> Hay circuitos que funcionan con lógica inversa, y en este caso se dice que una señal está activa cuando vale 0. En este curso, sin embargo, cuando decimos que una señal está activa queremos decir que vale 1.

Puesto que las señales sólo pueden tomar dos valores, diremos que uno es el contrario del otro. Así pues, podemos afirmar que cuando una señal no vale 1, entonces seguro que vale 0, y viceversa.

Tomamos un circuito que tenga sólo una señal de entrada (que llamamos *x*), un dispositivo electrónico y una señal de salida (que llamamos *z*).

Dado que tanto *x* como *z* sólo pueden valer 0 ó 1, sólo existen cuatro dispositivos electrónicos diferentes que pueden interconectar *x* y *z*:

- Un dispositivo que haga que la salida z valga siempre 0 (este dispositivo consistiría en conectar la salida con una fuente de tensión de 0 voltios).

- Un dispositivo que haga que la salida valga siempre lo mismo que la entrada *x* (este dispositivo consistiría en conectar directamente la salida con la entrada).

- Un dispositivo que haga que la salida valga siempre lo contrario de lo que vale la entrada (este dispositivo consistiría en un inversor del nivel de tensión).

- Un dispositivo que haga que la salida valga siempre 1 (este dispositivo consistiría en conectar la salida directamente con la tensión de alimentación).

En otras palabras, podemos decir que sólo hay cuatro funciones lógicas que tengan una sola variable de entrada, tal como se muestra en la figura 1:

Figura 1

| Función lógica | Descripción | Valor de la función cuando x = 0 | Valor de la función cuando x = 1 |
|---|---|---|---|
| $z = f_0(x) = 0$ | función constante 0 | 0 | 0 |
| $z = f_1(x) = x$ | función identidad | 0 | 1 |
| $z = f_2(x) = x'$ | función contrario | 1 | 0 |
| $z = f_3(x) = 1$ | función constante 1 | 1 | 1 |

**Nota**

Introducimos aquí la nomenclatura *x'* para referirnos a la función "contrario de *x*". Más adelante la definiremos con más propiedad.

Tomamos ahora un circuito que tenga dos señales de entrada (que llamamos *x* e *y*), un dispositivo electrónico y una señal de salida (que llamamos *z*).

En este caso, existen dieciséis dispositivos electrónicos diferentes que puedan interconectar las entradas con la salida, que corresponden a las funciones lógicas que se muestran en la figura 2.

Figura 2

| Función lógica | Valor de la función cuando | | | |
|---|---|---|---|---|
| | x = 0, y = 0 | x = 0, y = 1 | x = 1, y = 0 | x = 1, y = 1 |
| $z = g_0(x,y)$ | 0 | 0 | 0 | 0 |
| $z = g_1(x,y)$ | 0 | 0 | 0 | 1 |
| $z = g_2(x,y)$ | 0 | 0 | 1 | 0 |
| $z = g_3(x,y)$ | 0 | 0 | 1 | 1 |
| $z = g_4(x,y)$ | 0 | 1 | 0 | 0 |
| $z = g_5(x,y)$ | 0 | 1 | 0 | 1 |
| $z = g_6(x,y)$ | 0 | 1 | 1 | 0 |
| $z = g_7(x,y)$ | 0 | 1 | 1 | 1 |
| $z = g_8(x,y)$ | 1 | 0 | 0 | 0 |
| $z = g_9(x,y)$ | 1 | 0 | 0 | 1 |
| $z = g_{10}(x,y)$ | 1 | 0 | 1 | 0 |
| $z = g_{11}(x,y)$ | 1 | 0 | 1 | 1 |
| $z = g_{12}(x,y)$ | 1 | 1 | 0 | 0 |
| $z = g_{13}(x,y)$ | 1 | 1 | 0 | 1 |
| $z = g_{14}(x,y)$ | 1 | 1 | 1 | 0 |
| $z = g_{15}(x,y)$ | 1 | 1 | 1 | 1 |

Así pues, si tenemos en cuenta los circuitos con una o dos entradas, podemos llegar a diseñar hasta 4 + 16 = 20 dispositivos electrónicos diferentes. Ahora bien, en la práctica sólo se construye un subconjunto de éstos, concretamente los que corresponden a las funciones $f_2$, $g_1$ y $g_7$ (y otras que veremos más adelante), porque a partir de estas funciones se pueden construir todas las demás, tal como veremos en el apartado 1.2.

A continuación, veremos que existe una correspondencia entre los elementos de un circuito lógico y la lógica intuitiva (de aquí deriva la denominación de circuitos "lógicos"). La lógica tiene cinco componentes básicos:

- Los valores *falso* y *cierto*.
- Las conjunciones *y* y *o*.
- La partícula de negación *no*.

Por ejemplo, sean las dos frases siguientes:

frase_A: "Juan estudia química",
frase_B: "Carmen estudia piano".

Cada una de estas frases puede ser verdadera o falsa. A partir de éstas, de las conjunciones y la negación construimos ahora las tres frases siguientes:

frase_Y: "Juan estudia química **y** Carmen estudia piano",
frase_O: "Juan estudia química **o** Carmen estudia piano",
frase_NO: "Juan **no** estudia química".

Según la lógica:

- La frase_Y es verdadera sólo si son ciertas la frase_A y la frase_B, simultáneamente.
- La frase_O es verdadera si lo es la frase_A o bien la frase_B, o ambas.
- La frase_NO es verdadera sólo si la frase_A es falsa.

**Nota**

Normalmente, cuando utilizamos la conjunción
o en lenguaje natural lo hacemos de manera exclusiva. Es decir, si decimos "X o Y" nos referimos al hecho de que o bien es cierto X o bien lo es Y, pero no ambas a la vez. La *o* lógica, en cambio, incluye también la posibilidad de que las dos afirmaciones sean ciertas.

La correspondencia de la lógica con los elementos de un circuito lógico es la siguiente:

| Lógica | Circuitos lógicos |
|---|---|
| falso | 0 |
| cierto | 1 |
| conjunción *y* | función $g_1$ |
| conjunción *o* | función $g_7$ |
| partícula *no* | función $f_2$ |

En efecto, sean dos señales lógicas x e y. Si analizamos las tablas de las figuras 1 y 2, podemos observar lo siguiente:

- $g_1(x,y)$ vale 1 si valen 1 las dos variables $x$ e $y$ simultáneamente;
- $g_7(x,y)$ vale 1 si $x$ vale 1 o bien $y$ vale 1 o bien las dos valen 1;
- $f_2(x)$ vale 1 sólo si $x$ vale 0.

Es decir, se cumple lo mismo que en el ejemplo de las frases. Por eso, la función $g_1$ se llama AND (la conjunción *y* en inglés), la función $g_7$ se denomina OR (la conjunción *o* en inglés) y la función $f_2$ se llama NOT (*no* en inglés).

Así pues, los circuitos lógicos se construyen a partir del mismo fundamento que la lógica. En el siguiente apartado se estudia este fundamento.

## 1.2. Álgebra de Boole

Un **álgebra de Boole** es una entidad matemática formada por un conjunto que contiene dos elementos, unas operaciones básicas sobre estos elementos y una lista de axiomas que definen las propiedades que cumplen las operaciones.

**George Boole**

Formalizó matemáticamente la lógica en 1854, cuando definió lo que hoy se conoce como *álgebra de Boole*. De hecho, aquí presentamos un caso particular de álgebra de Boole, denominado *álgebra de Boole binaria*; en general, el conjunto de un álgebra de Boole puede tener más de dos elementos.

En 1938, Claude Shannon definió el comportamiento de los circuitos lógicos sobre el fundamento del álgebra de Boole y creó el *álgebra de conmutación*.

Los dos **elementos** de un álgebra de Boole se pueden llamar *falso* y *cierto* o, más usualmente, 0 y 1. De este modo, una variable booleana o **variable lógica** puede tomar los valores 0 y 1.

Las **operaciones booleanas** básicas son las siguientes:

- La **negación** o complementación o NOT, que corresponde a la partícula *no* y se representa con una comilla simple (′). Así, la expresión $x'$ denota la negación de la variable $x$, y se lee "*no x*".

- El **producto lógico** o **AND**, que corresponde a la conjunción *y* de la lógica y se representa con el símbolo "·". Así, si $x$ e $y$ son variables lógicas, la expresión $x \cdot y$ denota su producto lógico y se lee "*x* e *y*".

- La **suma lógica** u **OR**, que corresponde a la conjunción *o* y se representa con el símbolo "+". Así, la expresión "$x + y$" denota la suma lógica de las variables $x$ e $y$, y se lee "*x* o *y*".

La operación de negación se puede representar también de otras maneras. Por ejemplo: $!x$ o, más usualmente, $\bar{x}$ .

Estas operaciones booleanas básicas se pueden definir escribiendo el resultado que dan para cada posible combinación de valores de las variables de entrada, tal como se muestra en la figura 3. En las tablas de esta figura aparecen a la izquierda de la línea vertical todas las combinaciones posibles de las variables de entrada, y a la derecha, el resultado de la operación para cada combinación. Podemos comprobar que corresponden a las funciones $f_2$, $g_1$ y $g_7$ que hemos visto en el apartado anterior (figuras 1 y 2).

**Atención**

Es importante no confundir los operadores · y + con las operaciones *producto entero* y *suma entera* a las que estamos acostumbrados. El significado de los símbolos vendrá determinado por el contexto en el que nos encontremos. Así, si trabajamos con enteros, 1 + 1 = 2 ("uno más uno igual a dos"), mientras que si estamos en un contexto booleano, 1 + 1 = 1 ("cierto o cierto igual a cierto", tal como se puede ver en la tabla de la operación OR).

Figura 3

| x | $x'$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Operación NOT

| $x$ | $y$ | $x \cdot y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Operación AND

| $x$ | $y$ | $x + y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Operación OR

Los **axiomas** que describen el comportamiento de las operaciones booleanas son los siguientes:

La formulación que se presenta aquí de los axiomas booleanos fue introducida por Huntington en 1904, a partir de la descripción original de Boole.

Sean $x$, $y$ y $z$ variables lógicas. Entonces se cumple lo siguiente:

**a)** La **propiedad conmutativa:**

$$x + y = y + x$$
$$x \cdot y = y \cdot x$$

**b)** La **propiedad asociativa:**

$$x + (y + z) = (x + y) + z$$
$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

c) La **propiedad distributiva:**

$$x \cdot (y + z) = x \cdot y + x \cdot z$$
$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

Observamos que la segunda igualdad de la propiedad distributiva no se cumple en el contexto de la aritmética entera.

d) Los **elementos neutros:**

$$x + 0 = x$$
$$x \cdot 1 = x$$

e) La **complementación**. Para cualquier $x$ se cumple lo siguiente:

$$x + x' = 1$$
$$x \cdot x' = 0$$

A partir de estos axiomas, se puede demostrar una serie de leyes o teoremas muy útiles para trabajar con expresiones algebraicas booleanas.

**Teoremas del álgebra de Boole**

Si $x$, $y$ y $z$ son variables lógicas, se cumplen las siguientes leyes:

**1) Principio de dualidad**

Toda identidad deducida a partir de los axiomas continúa siendo cierta si las operaciones + y · y los elementos 0 y 1 se intercambian en toda la expresión.

Esta ley se puede comprobar, por ejemplo, examinando las parejas de expresiones que aparecen en la definición de los axiomas.

**2) Ley de idempotencia**

$$x + x = x$$
$$x \cdot x = x$$

**3) Ley de absorción**

$$x + x \cdot y = x$$
$$x \cdot (x + y) = x$$

**4) Ley de dominancia**

$$x + 1 = 1$$
$$x \cdot 0 = 0$$

**5) Ley de involución**

$$(x')' = x$$

**6) Leyes de De Morgan**

$$(x + y)' = x' \cdot y'$$
$$(x \cdot y)' = x' + y'$$

Un buen ejercicio para comprobar que se cumplen estas leyes es pensar en su correspondencia con la lógica, tal y como se hace en la última parte del apartado 1.1.

En las expresiones algebraicas utilizamos los paréntesis de la misma forma que estamos acostumbrados a hacerlo en aritmética entera; por ejemplo, la expresión $x + y \cdot z$ es lo mismo que la $x + (y \cdot z)$ y es diferente que $(x + y) \cdot z$.

Para negar una expresión entera la pondremos entre paréntesis y, por tanto, $x + y'$ es diferente de $(x + y)'$.

**Actividades**

**1.** Evaluad el valor de la expresión $x + y \cdot (z + x')$ para los casos:
$[x\ y\ z] = [0\ 1\ 0]$,
$[x\ y\ z] = [1\ 1\ 0]$,
$[x\ y\ z] = [0\ 1\ 1]$.

**2.** Demostrad las leyes de De Morgan de acuerdo con todos los posibles valores que pueden tomar las variables que aparecen.

**3.** Demostrad la ley 4 del álgebra de Boole a partir de la ley 2 y del axioma *c* y *e*.

## 1.3. Representación de funciones lógicas

Las funciones lógicas se pueden expresar de varias maneras. Las que usaremos nosotros son las **expresiones algebraicas** y las **tablas de verdad**.

### 1.3.1. Expresiones algebraicas

Las **expresiones algebraicas** están formadas por variables lógicas, los elementos 0 y 1, los operadores producto (·), suma (+) y negación (′), y los símbolos (,) e =.

Mediante estos elementos se puede expresar cualquier función lógica. Por ejemplo, la función $g_4$ de la figura 2 se puede expresar así:

$$g_4(x,y) = x' \cdot y$$

La expresión $x' \cdot y$ vale 1 (es cierta) sólo si valen 1 (son ciertas) las subexpresiones $x'$ e $y$ simultáneamente. La expresión $x'$ vale 1 sólo si $x$ vale 0. En la descripción de la función $g_4$ (figura 2) se puede comprobar que sólo vale 1 en el caso en que $x = 0$ e $y = 1$.

Una misma función lógica se puede expresar mediante infinitas expresiones algebraicas equivalentes.

Para saber si dos expresiones algebraicas son equivalentes (es decir, expresan la misma función) podemos analizar si una se puede derivar de la otra usando los axiomas y leyes del álgebra de Boole.

Por ejemplo, la primera ley de De Morgan nos dice que las funciones lógicas $f$ y $g$ son la misma:

$$f(x, y) = (x + y)',$$
$$g(x, y) = x' \cdot y'.$$

A la inversa, a partir de la expresión algebraica de una función podemos encontrar otras expresiones equivalentes aplicándole los axiomas y leyes del álgebra de Boole.

Por ejemplo, si tenemos la siguiente función:

$$f(x, y, z) = x \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y \cdot z,$$

y aplicamos el axioma $c$ (propiedad distributiva), obtenemos la siguiente expresión equivalente:

$$f(x, y, z) = x \cdot y' \cdot z + (x' + x) \cdot y \cdot z.$$

Por el axioma $e$ (de complementación) y después el $d$ (de elementos neutros), obtenemos lo que expresamos a continuación:

$$f(x, y, z) = x \cdot y' \cdot z + 1 \cdot y \cdot z = x \cdot y' \cdot z + y \cdot z.$$

Por la propiedad distributiva,

$$f(x, y, z) = (x \cdot y' + y) \cdot z.$$

Y, de este modo, podríamos seguir encontrando infinitas expresiones equivalentes para la misma función.

Al igual que cuando trabajamos con ecuaciones, podemos omitir el signo "·" en los productos lógicos para simplificar la escritura de las expresiones algebraicas: si escribimos seguidos los nombres de diferentes variables, sobreentenderemos que el producto lógico se obtiene de éstas.

Así, las dos expresiones siguientes son igualmente válidas:

$$x_1' \cdot x_0 + x_2 \cdot x_1' \cdot x_0', \text{ o bien}$$
$$x_1'x_0 + x_2x_1'x_0'.$$

### Actividades

**4.** Encontrad una expresión algebraica para las funciones $g_1$, $g_3$, $g_6$, $g_7$ y $g_{10}$ de la figura 2.

**5.** Encontrad una expresión más sencilla para esta función: $f = wx + xy' + yz + xz' + xy$.

**6.** Sea una función de tres variables $x$, $y$ y $z$. Encontrad una expresión algebraica de la misma suponiendo que la función debe valer 1 cuando se cumpla alguna de las siguientes condiciones:
- $x = 1$ o $y = 0$,
- $x = 0$ y $z = 1$,
- las tres variables valen 1.

**7.** Se dispone de dos cajas fuertes electrónicas, $A$ y $B$. Cada una de éstas tiene una señal asociada, $x_A$ y $x_B$ respectivamente, que vale 1 cuando la caja está abierta y 0 cuando está cerrada. Se tiene también un interruptor general que tiene una señal asociada $i_g$, que vale 1 si el interruptor está cerrado y 0 si no lo está.
Se quiere construir un sistema de alarma antirrobos, que generará una señal de salida $s$. Esta señal tiene que valer 1 cuando alguna caja fuerte esté abierta y el interruptor esté cerrado.
Escribid la expresión algebraica de la función $s = f(x_A, x_B, i_g)$.

**8.** Juan se ha examinado de tres asignaturas. Sus amigos han visto los resultados de los exámenes y le han comentado lo siguiente:
- Has aprobado matemáticas o física, dice el primero.
- Has suspendido química o matemáticas, dice el segundo.
- Has aprobado sólo dos asignaturas, dice el tercero.

Entendemos que la "o" de estas frases es exclusiva. Es decir, la primera frase se podría sustituir por "o bien has aprobado matemáticas y has suspendido física, o bien has suspendido matemáticas y has aprobado física", y de manera similar con la segunda frase.
**a)** Escribid las expresiones algebraicas de las afirmaciones de cada uno de los amigos.
**b)** Utilizando los axiomas y teoremas del álgebra de Boole, deducid qué asignaturas ha aprobado Juan y cuál ha suspendido.

### 1.3.2. Tablas de verdad

Una **tabla de verdad** expresa una función lógica especificando el valor que tiene la función para cada posible combinación de valores de las variables de entrada.

**Tabla de verdad**

En aritmética entera es imposible escribir la tabla de verdad de una función, ya que las variables de entrada pueden tomar infinitos valores y, por tanto, la lista debería ser infinita. En álgebra de Boole es posible gracias al hecho de que las variables pueden adoptar sólo dos valores.

Concretamente, a la izquierda de la tabla hay una lista con todas las combinaciones de valores posibles de las variables de entrada y, a la derecha, el valor de la función para cada una de las combinaciones. Por ejemplo, las tablas que habíamos visto en la figura 3 eran, de hecho, las tablas de verdad de las funciones NOT, AND y OR.

Si una función tiene *n* variables de entrada, y dado que una variable puede tomar sólo dos valores, 0 ó 1, las entradas pueden adoptar $2^n$ combinaciones de valores diferentes. Por tanto, su tabla de verdad tendrá a la izquierda *n* columnas (una para cada variable) y $2^n$ filas (una para cada combinación posible). A la derecha habrá una columna con los valores de la función.

Las filas se escriben siempre en *orden lexicográfico*, es decir, si interpretamos las diferentes combinaciones como números naturales, escribiremos primero la combinación correspondiente al 0 (formada sólo por ceros), después la correspondiente al 1 y así sucesivamente, en orden creciente hasta la correspondiente al $2^n - 1$ (formada sólo por unos).

**Nota**

No obstante las combinaciones se podrían escribir en cualquier orden. Por ejemplo, las dos tablas siguientes expresan la misma función:

| *x* | *y* | *f* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| *x* | *y* | *f* |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |

La figura 4 muestra la estructura de las tablas de verdad para funciones de una, dos y tres variables de entrada.

Figura 4

Estructura de la tabla de la verdad de una función de

1 variable

| *a* | *f* |
|---|---|
| 0 | |
| 1 | |

2 variables

| *a* | *b* | *f* |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

3 variables

| *a* | *b* | *c* | *f* |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | 0 | 1 | |
| 0 | 1 | 0 | |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | |
| 1 | 0 | 1 | |
| 1 | 1 | 0 | |
| 1 | 1 | 1 | |

Por ejemplo la función $g_5$ de la figura 2 tiene la siguiente tabla de verdad:

| $x$ | $y$ | $g_5$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Es importante notar que el orden en que ponemos las columnas de las variables es significativo. Por ejemplo, podíamos haber escrito la tabla de verdad de la función $g_5$ de la siguiente manera:

| $y$ | $x$ | $g_5$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Es bastante usual denominar las variables de una función con una misma letra y diferentes subíndices; por ejemplo, $x_2$, $x_1$, $x_0$. Por convención, pondremos en la columna situada más a la izquierda la variable de subíndice más alto y, en la que se encuentra más a la derecha, la de subíndice más bajo.

Diremos que la variable que ponemos en la columna más a la izquierda en una tabla de verdad es la variable **de más peso**, y la que ponemos en la columna más a la derecha es la **de menos peso.** Una vez fijado el orden de las columnas y las filas, la tabla de verdad de una función es **única**.

Podemos expresar el comportamiento de diferentes funciones que tienen las mismas variables de entrada en una única tabla de verdad. En este caso, a la derecha de la línea vertical habrá tantas columnas como funciones. No estableceremos ninguna convención para ordenar las columnas correspondientes a las funciones (se pueden poner en cualquier orden). A continuación se muestra un ejemplo.

| $x_2$ | $y_1$ | $x_0$ | $f_0$ | $f_1$ | $f_2$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 |

**Actividades**

**9.** Escribid la tabla de verdad de las funciones $g_1$, $g_3$, $g_6$, $g_7$ y $g_{10}$ de la figura 2.

**10.** Dibujad la estructura de la tabla de verdad de la función $f(x_3, x_2, x_1, x_0)$.

**11.** Especificad la tabla de verdad de una función de cuatro variables, $x_3$, $x_2$, $x_1$ y $x_0$, que vale 1 sólo cuando un número par de las variables valen 1 (recordad que el 0 es un número par).

### 1.3.3. Correspondencia entre expresiones algebraicas y tablas de verdad

Es importante saber pasar con agilidad de una expresión algebraica de una función a su tabla de verdad, y viceversa.

**Para obtener la tabla de verdad a partir de una expresión algebraica** podemos actuar de dos maneras:

**a)** Evaluar la expresión para cada combinación de las variables y escribir en el lugar correspondiente de la tabla el resultado de la evaluación.

Por ejemplo, con la función siguiente:

$$f(a, b, c) = a + a' b' c + ac'.$$

Si evaluamos la expresión para el caso de la combinación $[a\ b\ c] = [0\ 0\ 0]$, obtenemos el siguiente resultado:

$$0 + 1 \cdot 1 \cdot 0 + 0 \cdot 1 = 0 + 0 + 0 = 0.$$

Por tanto, en la primera fila de la tabla irá un 0.

Para la combinación $[a\ b\ c] = [0\ 0\ 1]$ obtenemos el resultado que presentamos a continuación:

$$0 + 1 \cdot 1 \cdot 1 + 0 \cdot 0 = 0 + 1 + 0 = 1.$$

Por tanto, en la segunda fila irá un 1. Y así sucesivamente hasta haber evaluado la función para todas las combinaciones.

**b)** Analizar la expresión *a vista*, deducir para qué valores de las variables de la función vale 1 ó 0, y rellenar la tabla.

Tomamos la misma función del ejemplo anterior. Vemos que siempre que $a = 1$, la función vale 1 (por la ley 4, de dominancia). Por tanto, podemos poner un 1 en todas las filas en las que $a = 1$ (las cuatro últimas).

A continuación estudiamos los casos en que $a = 0$ (los que nos faltan por rellenar). La expresión de la función nos queda de la siguiente manera:

$$f(0, b, c) = 0 + 0' \cdot b' \cdot c + 0 \cdot c' = b' \cdot c.$$

Esta expresión vale 1 sólo en el caso $[b\ c] = [0\ 1]$. Por tanto, pondremos un 1 en la fila correspondiente a la combinación $[0\ 0\ 1]$.

Para las combinaciones que aún nos quedan por rellenar, la función vale 0.

Observamos que el paso de expresión a tabla de verdad nos puede ser muy útil a la hora de determinar si dos expresiones algebraicas son equivalentes o no. Podemos obtener la tabla de verdad a partir de cada una de las expresiones, de modo que si las tablas resultantes son iguales, podemos concluir que las dos expresiones corresponden a una misma función.

### Actividades

**12.** Obtened la tabla de verdad de las siguientes funciones:
**a)** $f(x, y, z, w) = xy'zw' + yz'w + x'z + xyw + x'yz'w'$
**b)** $f(x, y, z, w) = xy + z$

**13.** Mediante tablas de verdad, estudiad si las dos expresiones siguientes son equivalentes:
**a)** $f = x'y' + yw + x'w$,
**b)** $g = x'y'z'w' + x'z'w + yz'w + xyw + x'z$.

**14.** Escribid la tabla de verdad correspondiente a la actividad 7.

**15.** Escribid la tabla de verdad correspondiente a la actividad 8. Encontrad la solución de la actividad a partir del estudio del contenido de esta tabla.

**El paso de tabla de verdad a expresión algebraica** se puede hacer "a vista", si tenemos suficiente experiencia. Sin embargo, será más cómodo si lo hacemos conociendo las expresiones en suma de mintérminos, que veremos en el siguiente apartado.

### 1.3.4. Expresiones en suma de mintérminos

Ya hemos visto que hay infinitas expresiones algebraicas equivalentes para una función cualquiera. Ahora bien, cualquier función lógica se puede expresar con una expresión en suma de productos. Es decir, una expresión de la forma siguiente:

$$A + B + C,$$

donde $A$, $B$ y $C$ son términos producto, es decir, tienen la forma que reproducimos a continuación.

$$X \cdot Y \cdot Z,$$

donde $X$, $Y$ y $Z$ son variables lógicas, bien negadas o bien sin negar. Por ejemplo, las dos expresiones siguientes corresponden a la misma función (lo podéis demostrar), pero sólo la segunda tiene la forma de suma de productos:

$$f(x, y, z) = (x + y\,z')'z,$$
$$f(x, y, z) = x'\,y'\,z + x'\,y\,z.$$

A partir de la tabla de verdad de una función es muy sencillo obtener una expresión en suma de productos. Veamos las tablas de verdad de las cuatro funciones siguientes:

| $x_2$ | $x_1$ | $x_0$ | $f_0$ | $f_1$ | $f_2$ | $F$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Si examinamos $f_0$, $f_1$ y $f_2$, observamos que tienen la particularidad de valer 1 sólo para una de las combinaciones de las variables de entrada, de manera que las podemos expresar fácilmente con un solo término producto, **en el que aparecen todas las variables**, negadas o sin negar. Las variables que valen 0 en la combinación que hace que la función valga 1 aparecerán negadas en el término producto; las que valen 1 aparecerán sin negar. Así, obtenemos los siguientes términos producto para las funciones anteriores:

$$f_0(x_2, x_1, x_0) = x_2'\, x_1'\, x_0.$$
$$f_1(x_2, x_1, x_0) = x_2 x_1' x_0'.$$
$$f_2(x_2, x_1, x_0) = x_2\, x_1\, x_0'.$$

> Se denomina *término mínimo* o **mintérmino** el término producto (único) que expresa una función que sólo vale 1 para una sola combinación de las variables de entrada.

**Nota**

En un mintérmino figuran siempre todas las variables de la función, ya sea negadas o sin negar.

No es tan sencillo obtener "a vista" una expresión para la función $F$, ya que vale 1 para varias combinaciones de las variables. Ahora bien, sí podemos ver fácilmente que $F$ vale 1 o bien $f_0$, o $f_1$ o $f_2$ valen 1. Es decir,

$$F = f_0 + f_1 + f_2.$$

Por tanto, obtenemos lo siguiente:

$$F(x_2,x_1,x_0) = x_2'\, x_1'\, x_0 + x_2\, x_1'\, x_0' + x_2\, x_1\, x_0',$$

que es una expresión algebraica de $F$ en forma de suma de productos.

> Así pues, a partir de la tabla de verdad de una función podemos obtener una expresión en suma de productos haciendo la suma lógica de los mintérminos que la forman. Por este motivo, la expresión que obtenemos de esta forma se llama **suma de mintérminos**.

**Nota**

Cualquier función lógica se puede expresar también en forma de producto de sumas, pero no trataremos este asunto en este curso.

Las expresiones en suma de mintérminos o producto de sumas se llaman *formas canónicas* de una función. Los mintérminos se denominan *términos canónicos*.

En el siguiente capítulo (apartado 2.3) veremos que a partir de una expresión en suma de mintérminos se puede obtener otra expresión en suma de productos más sencilla.

### Actividades

**16.** Obtened la expresión en suma de mintérminos de las funciones $f_0$, $f_1$, $f_2$ y $f_3$:

| $x_2$ | $x_1$ | $x_0$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**17.** Obtened la expresión en forma de suma de mintérminos de las siguientes funciones:
**a)** $f(x, y, z) = xy + z$,
**b)** $f(x, y, z, w) = x'y'zw + x'y'z' + xy'w + yz'w$.

### 1.4. Otras funciones comunes

Hemos visto que existen cuatro funciones lógicas de una variable (figura 1) y dieciséis de dos variables (figura 2). Nos hemos fijado, especialmente, en las funciones $f_2$, $g_1$ y $g_7$, porque corresponden a las operaciones básicas del álgebra de Boole, y las hemos llamado, respectivamente, *negación* (NOT), *producto lógico* (AND) y *suma lógica* (OR). A continuación hemos visto que cualquier función lógica se puede construir a partir de estas tres.

De entre las funciones de dos variables, existen otras que también reciben un nombre propio, ya que se utilizan de manera común. Son las siguientes:

**Función O-exclusiva o XOR**

Esta función vale 1 cuando alguna de las dos variables de entrada vale 1, pero no cuando valen 1 las dos a la vez; por eso recibe el nombre de O-*exclusiva*. Es la función $g_6$ de la figura 2.

Se representa con símbolo "⊕", y presenta esta tabla de verdad:

| $a$ | $b$ | $a \oplus b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$a \oplus b = a'b + ab'$

La función XOR de más de dos variables se calcula de este modo:

$$a \oplus b \oplus c \oplus d = (((a \oplus b) \oplus c) \oplus d).$$

Por tanto, da 0 si un número par de las variables vale 1, y da 1 si un número impar de las variables vale 1. Fijémonos que en el caso de dos variables, este hecho se traduce en lo siguiente:

$a \oplus b = 0 \Rightarrow a$ y $b$ tienen el mismo valor,
$a \oplus b = 1 \Rightarrow a$ y $b$ tienen valores diferentes.

Por tanto, la función XOR nos permite saber si dos variables son iguales o no lo son.

Otras propiedades de la XOR:

$$0 \oplus a = a,$$
$$1 \oplus a = a'.$$

Así, si hacemos la XOR de una variable con un 0, la variable queda inalterada, mientras que si hacemos la XOR con un 1, en la salida aparece la variable negada.

**Función NAND**

Es la negación de la AND, es decir:

$$a \text{ NAND } b = (a \cdot b)'.$$

Por tanto, vale 1 siempre que no se cumpla que las dos variables de entrada valgan 1. Es, pues, la función $g_{14}$ de la figura 2. Ésta es su tabla de verdad:

| *a* | *b* | (*a*+*b*)' |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Función NOR**

Es la negación de la OR, es decir:

$$a \text{ NOR } b = (a + b)'.$$

Por tanto, vale 1 sólo cuando ninguna de las dos variables de entrada vale 1. Es la función $g_8$ de la figura 2. Ésta es su tabla de verdad:

| *a* | *b* | (*a*+*b*)' |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

## 1.5. Funciones especificadas de manera incompleta

Existen funciones para las que algunas combinaciones de las variables de entrada no se producirán nunca. Por ejemplo, supongamos una función $f(x_2, x_1, x_0)$ en la que las variables corresponden a señales conectadas a tres aparatos de medición del sonido en una sala, de manera que tenemos lo siguiente:

$x_2$ = 1 si el sonido supera los 100 dB, $x_2$ = 0 de otro modo,
$x_1$ = 1 si el sonido supera los 200 dB, $x_1$ = 0 de otro modo,
$x_0$ = 1 si el sonido supera los 300 dB, $x_0$ = 0 de otro modo.

La combinación $[x_2, x_1, x_0]$ = [0 1 0] no se producirá nunca, ya que es imposible que el sonido esté por debajo de los 100 dB y por encima de los 200 dB simultáneamente. Tampoco se darán nunca las combinaciones [0 0 1], [0 1 1] y [1 0 1].

Las combinaciones que nunca se producirán se llaman **combinaciones "no importa"** o **combinaciones *don't care***.

En la tabla de verdad de una función escribiremos *x* en las filas correspondientes a las combinaciones "no importa". Por ejemplo, supongamos que la función anterior debe valer 1 si el sonido está entre 100 dB y 200 dB o si el sonido supera los 300 dB. Su tabla de verdad es la siguiente:

| $x_2$ | $x_1$ | $x_0$ | $f_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | x |
| 0 | 1 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | x |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

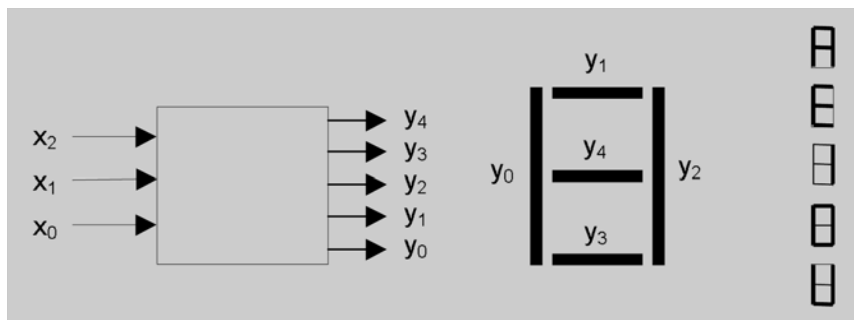
En el capítulo siguiente veremos cómo se pueden obtener expresiones algebraicas de funciones especificadas de manera incompleta.

### Actividades

**18.** Se quiere construir un sistema que compute cinco funciones de salida ($y_1$, $y_2$, $y_3$, $y_4$ e $y_5$) de tres variables ($x_2$, $x_1$ y $x_0$). Estas tres variables codifican las cinco vocales, tal como se muestra en la siguiente tabla:

| vocal | $x_2$ | $x_1$ | $x_0$ |
|---|---|---|---|
| A | 0 | 0 | 0 |
| E | 0 | 0 | 1 |
| I | 0 | 1 | 0 |
| O | 0 | 1 | 1 |
| U | 1 | 0 | 0 |

Cada una de las cinco funciones está asociada a un segmento de un *visualizador de cinco segmentos* como el que se muestra en la figura. Por ejemplo, cuando $y_1$ vale 1, el segmento que hay encima del visualizador se ilumina. La figura muestra también qué segmentos se deben iluminar para formar las diferentes vocales.




En cada momento el visualizador debe formar la vocal que codifiquen las variables de entrada en ese momento. Escribid la tabla de verdad de las cinco funciones.

**19.** Se quiere diseñar un sistema de riego de una planta con control de la temperatura y de la humedad de la tierra. El sistema tiene tres señales de entrada (variables) y dos de salida (funciones).

Entradas:

– Un sensor de temperatura (*t*): se pone a 1 si la temperatura de la tierra supera un límite prefijado *T0*.

– Dos sensores de humedad de la tierra ($h_0$ y $h_1$): se ponen en 1 cuando la humedad de la tierra supera los límites *H0* y *H1*, respectivamente. El límite *H0* es inferior al límite *H1* (*H0* < *H1*).

Salidas:

– *Regar* (*R*): cuando se pone en 1 se activa el riego de la planta.
– *Calentar* (*E*): cuando se pone en 1 se activa el calentamiento de la tierra.

Las especificaciones del sistema son las siguientes:

– La planta se riega siempre que la tierra está seca, es decir, siempre que no supera el límite *H0*.
– También se riega cuando la temperatura supera el límite *T0* y la humedad de la tierra es inferior a *H1*.
– La tierra de la planta se calienta cuando la temperatura es inferior a *T0* y la humedad superior a *H0*.

Escribid la tabla de verdad de las funciones *R* y *E*.

# 2. Implementación de circuitos lógicos combinacionales

## 2.1. Puertas lógicas. Síntesis y análisis

Ved las figuras 1 y 2 en el subapartado 1.1 de este módulo.

En las figuras 1 y 2 habíamos visto que se pueden construir hasta veinte dispositivos electrónicos diferentes para funciones de una y dos variables de entrada. Ahora bien, en la práctica sólo se construyen los que calculan las funciones NOT, AND, OR, XOR, NAND y NOR.

Los dispositivos electrónicos que calculan funciones lógicas se llaman puertas lógicas. Son dispositivos que están conectados a un cierto número de señales de entrada y a una señal de salida.

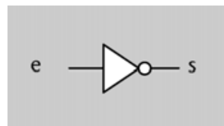
**Puertas lógicas**

Están formadas internamente por diferentes combinaciones de transistores, que son los dispositivos electrónicos más elementales.

A continuación presentamos el símbolo gráfico con el que se representa cada puerta lógica. En todas las figuras que aparecen a continuación, las señales de entrada quedan a la izquierda de las puertas y la señal de salida queda a la derecha.

**Puerta NOT o inversor**

Esta puerta se representa gráficamente con este símbolo:

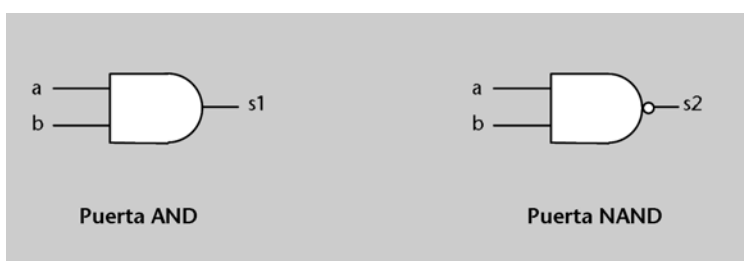



El funcionamiento es el siguiente: si en la entrada *e* hay un 0 (tensión baja), entonces en la salida *s* habrá un 1 (tensión alta). Y a la inversa, si hay un 1 en el punto *e*, entonces habrá un 0 en el punto *s*.

Por tanto, la puerta NOT implementa físicamente la función de negación: $s = e'$.

**Puertas AND y NAND**

Se representan gráficamente con estos símbolos:




**NAND**

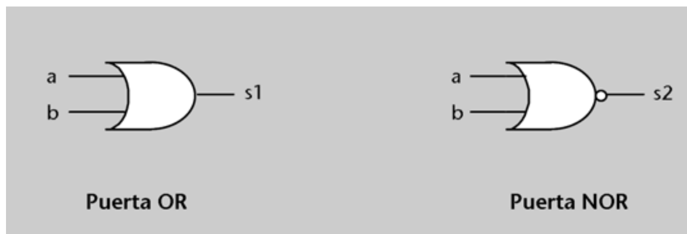
La función NAND es muy utilizada en el diseño de circuitos reales porque es muy fácil implementarla con transistores (es decir, no se implementa a partir de una puerta AND y una NOT).

La puerta AND implementa la función lógica AND, es decir, la salida $s1$ vale 1 sólo si las dos entradas $a$ y $b$ valen 1. Por tanto, $s1 = a \cdot b$.

La puerta NAND implementa la función lógica NAND. En el punto $s2$ hay un 1 siempre que en alguna de las dos entradas $a$ o $b$ haya un 0. Es decir, $s2 = (a \cdot b)'$.

**Puertas OR y NOR**

Se representan gráficamente con estos símbolos:




**El círculo en un circuito**

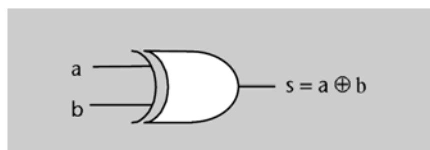
El círculo simboliza una negación en señales de salida (como en las puertas NOT, NAND y NOR). También se usa un círculo para negar señales de entrada, pero en este curso no utilizaremos esta posibilidad.

La puerta OR implementa la función lógica OR, es decir, en la salida $s1$ hay un 1 si cualquiera de las dos entradas está en 1. Por tanto, $s1 = a + b$.

La puerta NOR implementa la función lógica NOR. En el punto $s2$ encontraremos un 1 sólo cuando en las dos entradas haya un 0, es decir: $s2 = (a + b)'$.

**Puerta XOR**

Implementa la función lógica XOR, es decir, la salida $s$ vale 1 si alguna de las dos entradas vale 1, pero no si valen 1 las dos a la vez. Se representa gráficamente con este símbolo:




Todas las puertas (menos la NOT) pueden tener más de dos señales de entrada. Su funcionamiento es el siguiente:

- En una puerta AND de $n$ entradas, la salida vale 1 sólo cuando todas las $n$ entradas valen 1.
- En una puerta OR de $n$ entradas, la salida vale 1 cuando una o más de las $n$ entradas valen 1.
- En una puerta XOR de $n$ entradas, la salida vale 1 cuando hay un número impar de entradas en 1. El 0 se considera un número par y, por tanto, si todas las entradas valen 0, la salida también vale 0.

**Síntesis y análisis**

Cualquier función lógica se puede implementar usando estas puertas, es decir, se puede construir un circuito que se comporte como la función.

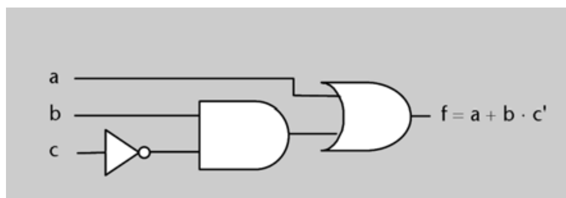
> El proceso de obtener el circuito que implementa una función a partir de una expresión algebraica se denomina **síntesis**.
>
> El proceso de obtener una expresión de una función a partir del circuito que la implementa se denomina **análisis**.

**Dibujo de las puertas**

En un circuito, las puertas se pueden dibujar en cualquier sentido: con la señal de salida a la derecha o a la izquierda, y también verticalmente con la señal de salida arriba o abajo, según convenga para la claridad del diseño.

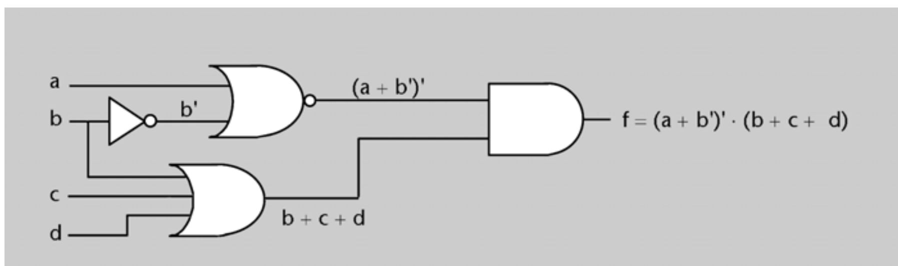
Para sintetizar (o implementar) una función a partir de su expresión algebraica, es suficiente con sustituir cada operador de la función por la puerta lógica adecuada. Por ejemplo, un término producto de tres variables se implementará con una puerta AND de tres entradas. Las puertas deben estar interconectadas entre sí y con las entradas tal como indica la expresión.

Por ejemplo, el circuito que implementa la función $f(a, b, c) = a + bc'$ es el siguiente:

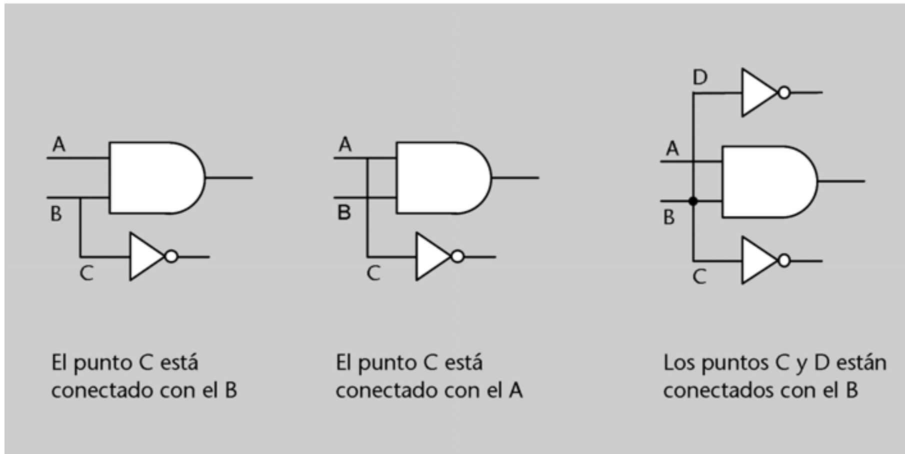



Para analizar un circuito, es necesario escribir las expresiones que corresponden a la salida de cada puerta, empezando desde las entradas del circuito hasta obtener la expresión correspondiente a la línea de salida del mismo.

Por ejemplo, el circuito siguiente implementa la función $f(a, b, c, d) = (a + b')'(b + c + d)$.



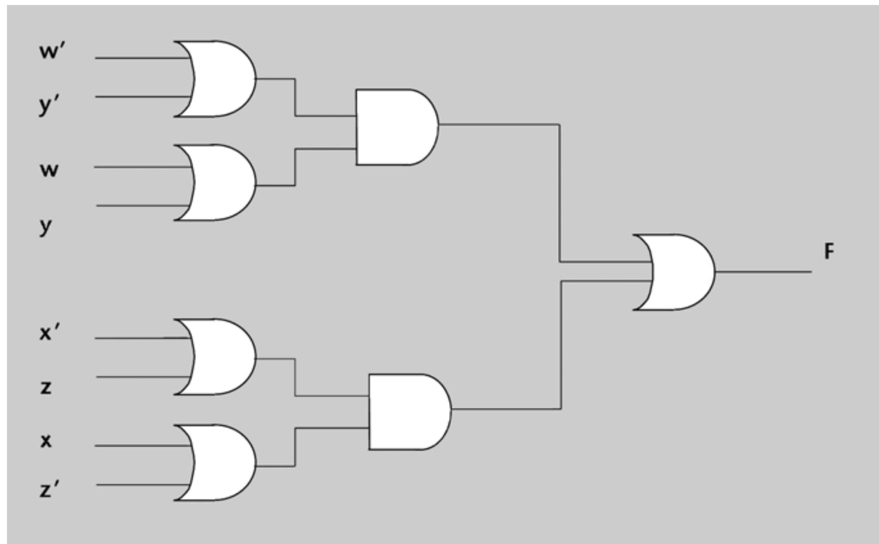

En un circuito, si una línea empieza sobre otra línea perpendicular, se entiende que las líneas están conectadas (y, por tanto, que tienen el mismo valor lógico). Si dos líneas perpendiculares se cruzan, se considera que no están conectadas. Si se pone un punto por encima de una línea, se interpreta que todas las líneas que lo tocan están conectadas. A continuación se muestran unos cuantos ejemplos.



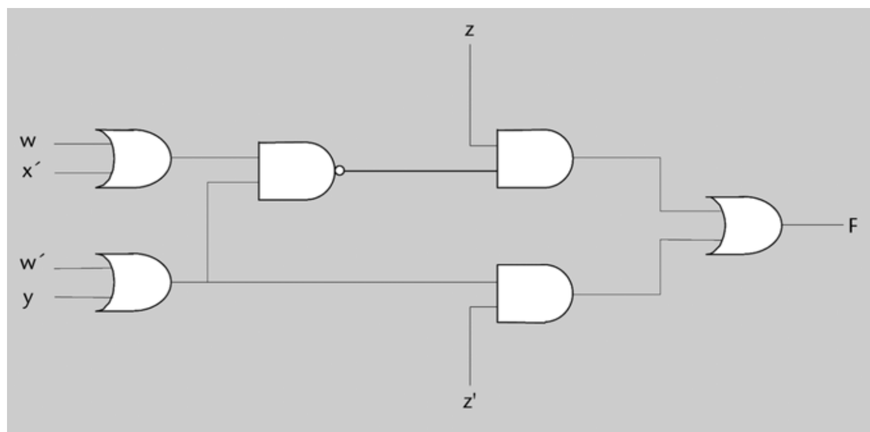

### Actividades

**20.** Analizad los siguientes circuitos:

**a)**




**b)**




**21.** Sintetizad la función $f(a, b, c, d) = d \cdot (a' + a \cdot b) \cdot (b' \oplus c)$.

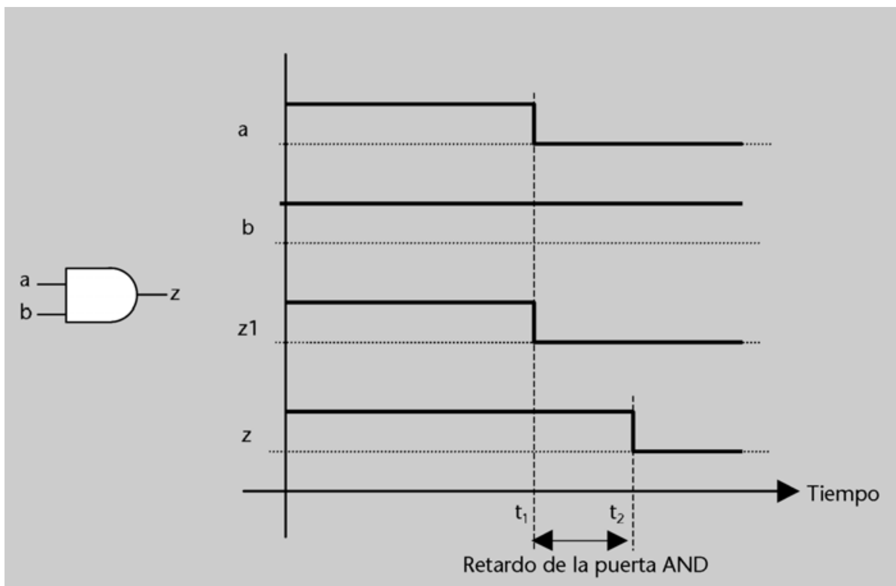
## 2.2. Diseño de circuitos a dos niveles

### 2.2.1. Retardos. Cronogramas. Niveles de puertas

Las puertas lógicas no responden instantáneamente a las variaciones en las señales de entrada, sino que experimentan un cierto **retardo**. La mejor forma de entender este concepto es observando la figura 5, en la que hay un circuito sencillo (sólo una puerta AND) y un **cronograma** de su funcionamiento.

Un **cronograma** es una representación gráfica de la evolución de las señales de un circuito a lo largo del tiempo.

Figura 5




**Cronograma**

Las líneas de puntos horizontales representan el valor lógico 0 para cada señal. Las líneas continuas gruesas representan el valor en que se encuentra cada señal en cada momento (0 si la línea continua está sobre la línea de puntos, 1 si está más arriba).

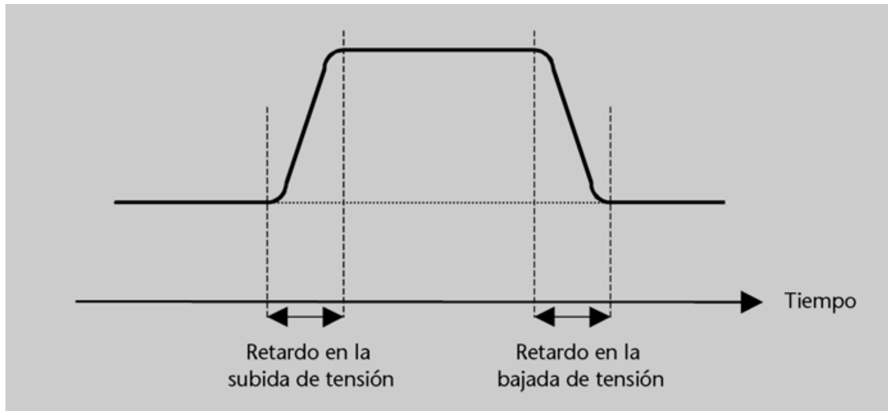
En vertical se pueden señalar con líneas discontinuas instantes determinados de tiempo (como $t_1$ y $t_2$ en el caso de la figura 5).

Supongamos que en las entradas *a* y *b* hay conectados dos interruptores que nos permiten en cada momento poner estas señales en 0 o en 1. En el ejemplo de la figura 5 hemos supuesto que, inicialmente, las dos señales están en 1 y que en el instante $t_1$ ponemos la señal *a* a 0. En este momento, la señal *z* se debería poner a 0, porque $0 \cdot 1 = 0$. Esta situación hipotética es la que se muestra en el cronograma con la señal *z*1. Sin embargo, en la realidad *z* no se pone a 0 hasta el instante $t_2$, a causa del hecho de que los dispositivos electrónicos internos de la puerta AND tardan un tiempo determinado en reaccionar. Diremos que la puerta AND tiene un retardo de $t_2 - t_1$.

Cada puerta tiene un retardo diferente que depende de la tecnología que se haya usado para construirla. Los retardos son muy pequeños, del orden de nanosegundos, pero es necesario tenerlos en cuenta a la hora de construir físicamente un circuito.

De hecho, la transición entre diferentes niveles de tensión de una señal tampoco es instantánea, sino que se produce tal como se muestra en la figura 6.

Figura 6




Uno de los objetivos de los ingenieros electrónicos que construyen circuitos es que el tiempo de respuesta del circuito sea lo más breve posible. Dado que cada puerta tiene un cierto retardo, un circuito será, en general, más rápido cuantos menos **niveles de puertas** haya entre las entradas y las salidas.

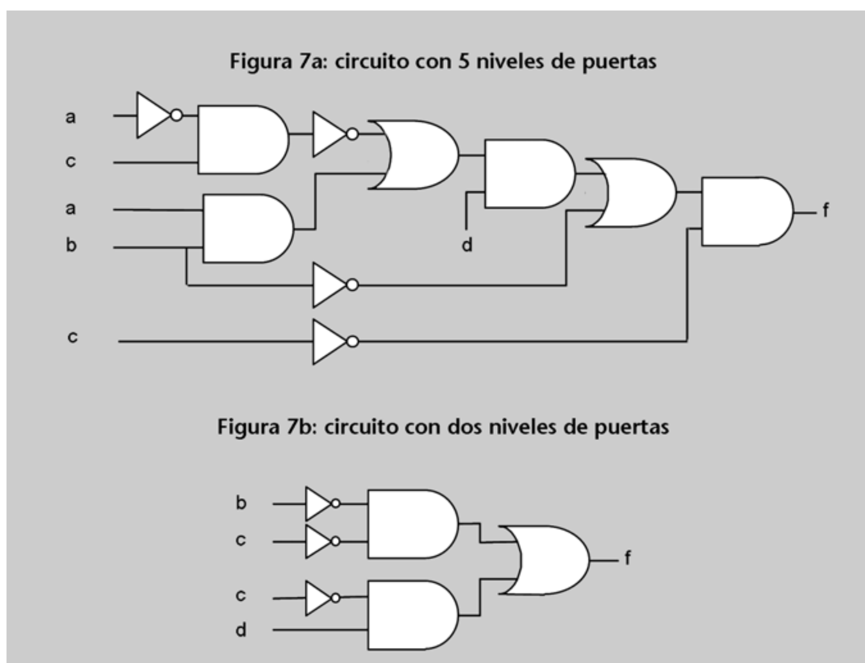
El tiempo de respuesta de una puerta depende, entre otras cosas, del número de entradas de la puerta.

El número de **niveles de puertas** de un circuito es el número máximo de puertas que una señal debe atravesar consecutivamente para generar la señal de salida.

Al contabilizar el número de niveles de puertas de un circuito **no se tienen en cuenta las puertas NOT** (por razones que no veremos en este curso).

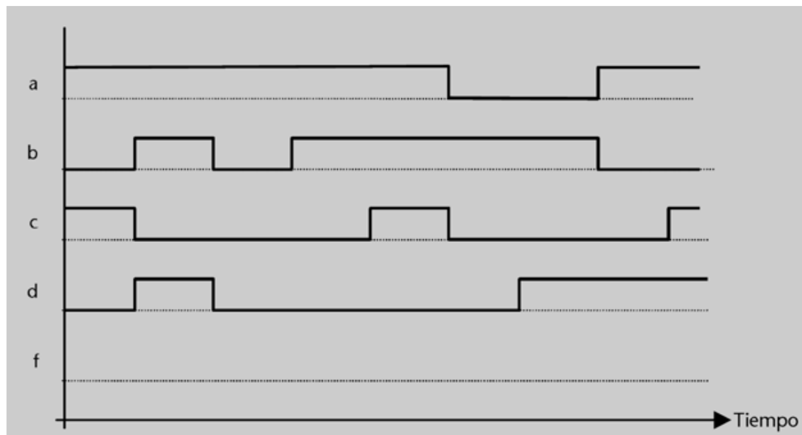
Por ejemplo, la figura 7 muestra el número de niveles de puertas de dos circuitos diferentes (podéis comprobar que la función que implementan es la misma). Un buen ingeniero elegiría el circuito de la figura 7*b* para implementar esta función, ya que es más rápido.

Figura 7



### Actividades

**22.** Completad el siguiente cronograma, suponiendo que la señal *f* corresponde a la función de la figura 7*b* y que las entradas toman los valores dibujados. Considerad que los retardos introducidos por las puertas son 0.

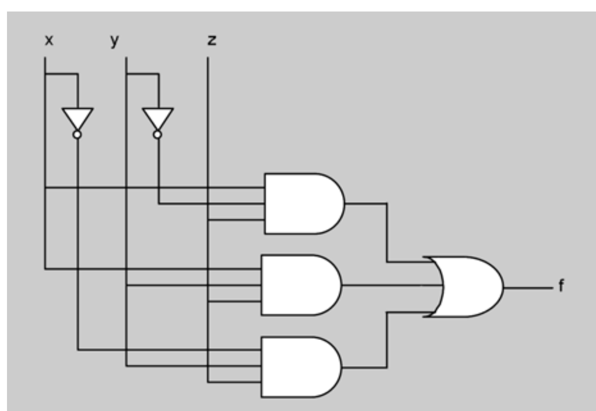



### 2.2.2. Síntesis a dos niveles

No existe una fórmula universal para encontrar la expresión de una función que dará lugar al circuito más rápido posible. Sin embargo, conocemos una forma de garantizar que un circuito no tenga más de dos niveles de puertas: partir de la expresión de la función en suma de mintérminos. En efecto, el circuito correspondiente a la suma de mintérminos tiene, además de las puertas NOT, un primer nivel de puertas AND (que computan los diferentes mintérminos) y un segundo nivel en el que habrá una única puerta OR, con tantas entradas como mintérminos tenga la expresión. Por ejemplo, la figura 8 muestra el circuito que se obtiene cuando se sintetiza esta función:

$$f = xy'z + xyz + x'yz.$$

Figura 8




Los circuitos que se obtienen a partir de las sumas de mintérminos se denominan **circuitos a dos niveles**. El proceso por el que los obtenemos se denomina **síntesis a dos niveles**.

Es posible que una función se implemente con un circuito que tenga menos de dos niveles. Sin embargo, no hay una manera de encontrarlo sistemáticamente. Así pues, las expresiones de las funciones en suma de mintérminos nos permiten construir los circuitos en dos niveles más rápidos posibles que podemos obtener de manera sistemática.

**Por otro lado, a partir de la expresión de una función en suma de mintérminos puede resultar un circuito con menos de dos niveles: si hay un sólo mintérmino no habrá el nivel OR.**

Tener en cuenta los retardos de las diferentes puertas y de las transiciones entre niveles de tensión es fundamental a la hora de construir circuitos. Sin embargo, en este curso consideraremos que los circuitos son ideales, de manera que no se tendrán nunca en cuenta los retardos, es decir, se asumirá que siempre son 0.

**Hay otros métodos de síntesis que generan circuitos con más de dos niveles pero más rápidos que los que se obtienen a partir de las expresiones de las funciones en suma de mintérminos. En este curso no los veremos.**

### Actividades

**23.** Haced la síntesis a dos niveles de las siguientes funciones:
**a)** $f(x,y,z,w) = x'y'z'w' + x'yz'w' + xy'zw' + xyzw + x'y'z'w + x'yzw$.
**b)** $f(x,y,z) = xz' + y'z + x'y$.

**24.** Se quiere diseñar un circuito combinacional que permita multiplicar dos números naturales de dos bits.
**a)** Indicad el número de bits de la salida.
**b)** Escribid la tabla de verdad de las funciones de salida.
**c)** Implementad el circuito a dos niveles.

**25.** Sintetizad la siguiente función a dos niveles:

| $x_2$ | $x_1$ | $x_0$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

## 2.3. Minimización de funciones

Ya hemos visto que uno de los objetivos que se tiene que conseguir cuando construimos un circuito es que sea tan rápido como sea posible. La síntesis a dos niveles nos permite conseguir de manera sistemática un grado de rapidez aceptable.

También es deseable que un circuito sea pequeño y barato. Estas dos características están relacionadas con el número de puertas del circuito: cuantas menos haya, más pequeño y barato será.

### 2.3.1. Simplificación de expresiones

En el capítulo anterior se ha visto cómo se puede obtener la expresión de una función en suma de mintérminos. A veces, estas expresiones se pueden

simplificar, lo que nos permitirá implementar la función con un circuito de menores dimensiones.

En concreto, las simplificaciones que nos serán útiles corresponden a los casos en que **dos o más mintérminos se pueden reducir a un único término producto**, en el que aparecerán menos variables.

Por ejemplo, sea la siguiente función:

$$f(x, y, z) = x'yz' + x'yz + \ldots$$

Estos dos mintérminos nos dicen que la función vale 1 si $x = 0$, $y = 1$ y $z = 0$ o bien si $x = 0$, $y = 1$ y $z = 1$. Sin embargo, esta información se puede resumir diciendo que la función vale 1 siempre que $x = 0$ e $y = 1$, independientemente del valor de $z$. Esto se puede expresar algebraicamente sacando el factor común en los dos mintérminos (es decir, aplicando las propiedades distributiva, de complementación y de elemento neutro):

$$f(x,y,z) = x'yz' + x'yz + \ldots =$$
$$= x'y(z'+z) + \ldots = x'y \cdot 1 + \ldots = x'y + \ldots$$

Tomamos ahora una función que vale 1 para las siguientes combinaciones [$x$ $y$ $z$] (entre otras):

[1 0 0]
[1 0 1]
[1 1 0]
[1 1 1]

Esta función vale 1 siempre que $x = 1$, independientemente del valor de $y$ y $z$. Aplicando el mismo razonamiento que antes, lo podemos expresar algebraicamente de este modo:

$$f(x,y,z) = xy'z' + xy'z + xyz' + xyz + \ldots =$$
$$= xy'(z'+z) + xy\,(z'+z) + \ldots =$$
$$= xy' + xy + \ldots = x\,(y'+y) + \ldots = x + \ldots$$

Así pues, vemos que podemos obtener un solo término producto en los siguientes casos:

**1.** Si en dos mintérminos todas las variables se mantienen constantes menos una, entonces podemos sacar factor común eliminando la variable que cam-

bia. En el término *producto resultante* aparecerán $n-1$ variables, siendo $n$ el número de variables de la función.

**2.** Si en cuatro mintérminos todas las variables se mantienen constantes menos dos, entonces podemos sacar factor común dos veces y eliminar las dos variables que cambian. En el término *producto resultante* aparecerán $n-2$ variables.

**También podemos detectar otros casos en los que se puede sacar factor común en una expresión algebraica. Por ejemplo, .**
**$f(x, y, z) = xy' + xz = x \cdot (y' + z)$**
**Sin embargo, en estos casos no nos interesan, ya que la expresión resultante no es una suma de productos.**

**3.** En general, si en $2^m$ mintérminos todas las variables se mantienen constantes menos m, entonces podemos sacar factor común $m$ veces y eliminar las $m$ variables que cambian. En el término *producto resultante* aparecerán $n-m$ variables.

El hecho de obtener la expresión en suma de productos más simplificada posible de una función, sacando factor común en los casos que se acaban de describir, se denomina **minimizar la función**.

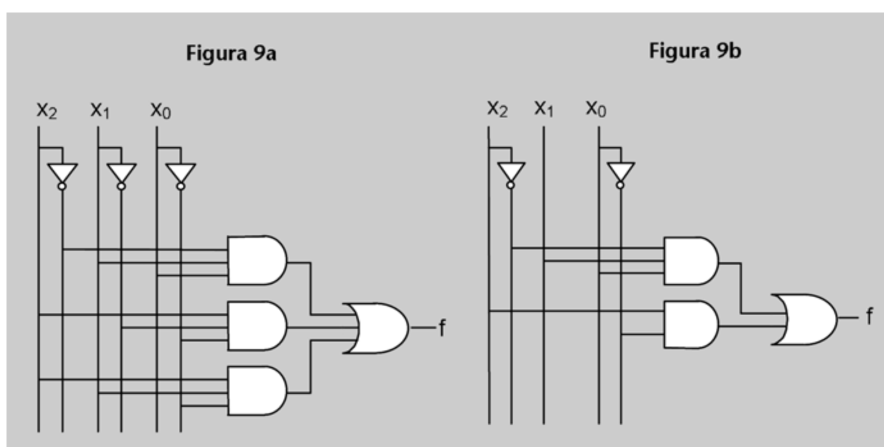
A partir de la expresión minimizada de una función, podremos construir circuitos de menores dimensiones y más baratos que los que obteníamos de la expresión en suma de mintérminos: quizá se necesitarán menos puertas NOT, algunas de las puertas AND serán de menos entradas, tendrán menos puertas AND y la puerta OR será de menos entradas.

La figura 9 muestra los circuitos correspondientes a la expresión en suma de mintérminos (figura 9*a*) y a la expresión minimizada (figura 9*b*) de esta función.

$$f(x_2,x_1,x_0) = x_2'x_1x_0 + x_2x_1'x_0' + x_2x_1x_0' =$$
$$= x_2'x_1x_0 + x_2x_0' (x_1'+x_1) = x_2'x_1x_0 + x_2x_0'.$$

Se puede observar que el circuito de la figura 9*a* tiene tres puertas NOT, tres AND de tres entradas y una OR de tres entradas. El circuito simplificado, en cambio, sólo tiene dos puertas NOT, dos AND (una de dos entradas y la otra de tres) y una puerta OR de dos entradas.

Figura 9



### 2.3.2. Síntesis mínima a dos niveles. Método de Karnaugh

Observando la tabla de verdad de la función podemos detectar los casos en que la expresión en suma de mintérminos se puede minimizar. Ahora bien, en algunos casos lo podemos ver fácilmente, pero en otros es más difícil. Por ejemplo, la figura 10*a* muestra la tabla de verdad de la función:

$$f(x_2, x_1, x_0) = x_2'x_1.$$

Sólo observando la tabla es fácil obtener esta expresión, porque las combinaciones por las que la función vale 1 (los mintérminos) se encuentran en filas consecutivas.

La función de la figura 10*b* es $f(x_2, x_1, x_0) = x_1'x_0$ porque vale 1 siempre que $x_1 = 0$ y $x_0 = 1$, independientemente de lo que valga $x_2$. Sin embargo, en este caso no es tan sencillo verlo a simple vista, ya que los mintérminos no están en filas consecutivas.

Figura 10

**Figura 10a**

| $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

**Figura 10b**

| $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

El **método de Karnaugh** proporciona una mecánica sencilla para detectar visualmente los casos en que se puede minimizar una expresión en suma de mintérminos. Por tanto, entre todos los circuitos a dos niveles que implementan una función, permite obtener fácilmente el menor de todos.

El método de Karnaugh consta de cuatro pasos:

**1)** Trasladar la tabla de verdad de la función a una estructura que se llama *mapa de Karnaugh*.

**2)** Detectar visualmente los casos en que se puede sacar factor común.

**3)** Deducir los términos producto más simples posible.

**4)** Obtener la expresión mínima de la función haciendo la suma lógica de los términos producto.

Veamos detalladamente cómo se lleva a cabo cada uno de estos procesos.

**1) Construcción del mapa de Karnaugh**

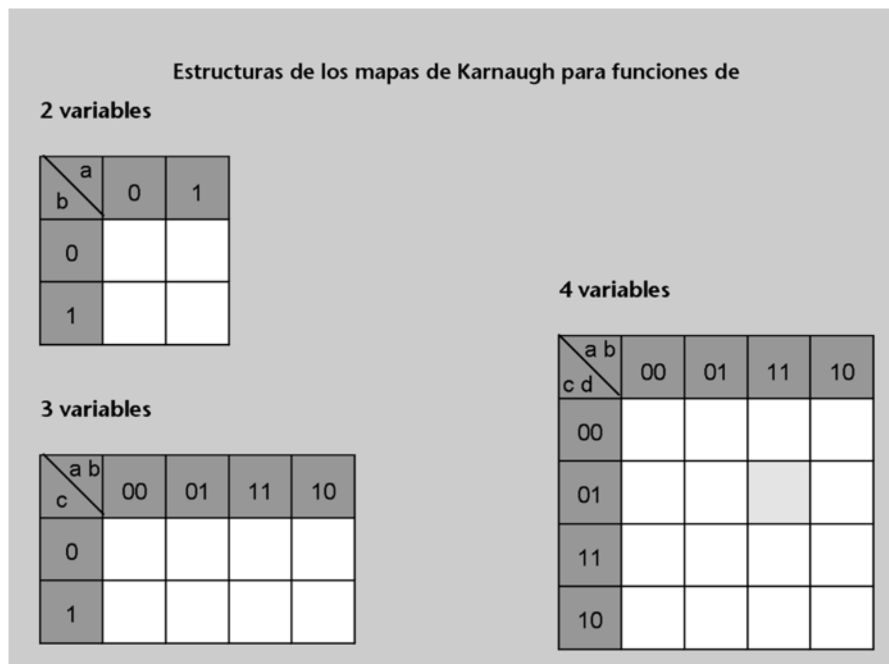
El **mapa de Karnaugh** es una transcripción de la tabla de verdad de una función a una estructura formada por casillas en la que cada una de éstas corresponde a una combinación de las variables (y, por tanto, a una fila de la tabla de verdad).

Se dice que dos casillas del mapa son **adyacentes** si corresponden a combinaciones en las que sólo cambia el valor de una variable.

La figura 11 muestra la estructura del mapa de Karnaugh para funciones de 2, 3 y 4 variables . Por ejemplo, la casilla que está sombreada con gris claro corresponde a la combinación [*a b c d* ] = [1 1 0 1].

**También es posible aplicar el método de Karnaugh a funciones de más de cuatro variables, pero en este curso no se estudiará. De hecho, para estos casos hay otros métodos más adecuados, que no estudiaremos.**

Figura 11




Fijémonos en que en las cabeceras de las filas y las columnas de los mapas de Karnaugh las combinaciones no están en orden lexicográfico.

De esta manera se cumple que las casillas adyacentes quedan dispuestas en el mapa de la siguiente manera:

**1.** Dos casillas donde únicamente cambia el valor de una variable son adyacentes.

**2.** En los mapas de tres y cuatro variables, las casillas de la columna más a la derecha también son adyacentes con las de la columna más a la izquierda.

**3.** En los mapas de cuatro variables, las casillas de la fila superior también son adyacentes con las de la fila inferior.

Una vez dibujado el mapa, pondremos dentro de cada casilla el valor de la función para la combinación correspondiente de variables, a partir de la tabla de verdad. En la figura 12 se puede ver un ejemplo de ello donde se muestra explícitamente la posición de algún mintérmino en la tabla.

Figura 12

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

| $x_1 x_0$ \ $x_3 x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 0 | 1 | 0 |
| 10 | 1 | 0 | 1 | 1 |

De hecho, es suficiente con rellenar las casillas para las que la función vale 1.

**2) Detección de los casos en que se puede sacar factor común**

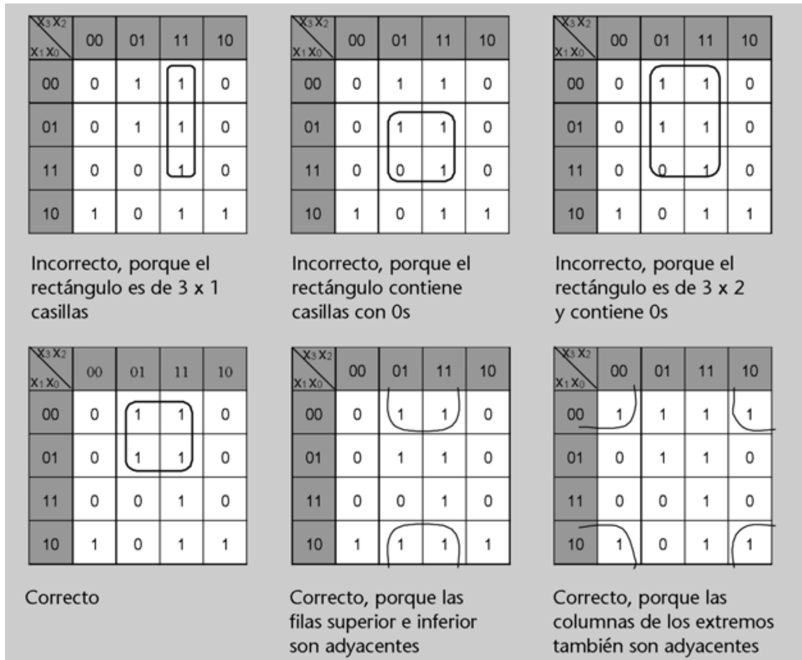
Supongamos que dos casillas adyacentes contienen unos; corresponden, pues, a dos mintérminos de la función. Sin embargo, entre éstas sólo cambia el valor de una variable (por la definición de adyacencia) y, por tanto, corresponden a un caso en el que se puede sacar factor común de los dos mintérminos.

Supongamos ahora que cuatro casillas adyacentes contienen unos. Entre éstas sólo cambia el valor de dos variables y, por tanto, corresponden a un caso en el que se puede sacar factor común dos veces.

Así, el segundo paso del método Karnaugh consiste en agrupar con rectángulos los unos que estén en casillas adyacentes, formando grupos de 1, 2, 4, 8 ó 16 unos. Los lados de estos rectángulos deben ser de un número de casillas potencia de 2, y en su interior sólo puede haber unos.

En la figura 13 se muestran varios ejemplos de rectángulos correctos e incorrectos.

Figura 13



Incorrecto, porque el rectángulo es de 3 x 1 casillas

Incorrecto, porque el rectángulo contiene casillas con 0s

Incorrecto, porque el rectángulo es de 3 x 2 y contiene 0s

Correcto

Correcto, porque las filas superior e inferior son adyacentes

Correcto, porque las columnas de los extremos también son adyacentes

La manera de agrupar los unos de un mapa no es única. Al hacer las agrupaciones, se debe tener en cuenta lo siguiente:

- Todos los unos deben formar parte de algún grupo.
- Los grupos tienen que ser lo más grandes posible (para que en cada término aparezca el menor número posible de variables). Por este motivo, empezaremos buscando los mayores grupos que podamos hacer (recordando que dentro de un grupo sólo puede haber unos).
- Cuantos menos grupos haya, mejor (para obtener el menor número posible de términos producto). El hecho de buscar los grupos más grandes reduce el número total de grupos que se harán.
- Un mismo 1 puede formar parte de más de un grupo si eso ayuda a satisfacer los dos objetivos anteriores. Si después de hacer los grupos grandes que veíamos a primera vista, queda algún 1 disperso, vemos si lo podemos agrupar con otros unos que ya formen parte de algún grupo.

En la figura 14 se muestran dos formas de agrupar los unos del mapa del ejemplo anterior. La de la figura 14*a* no es incorrecta, pero la de la figura 14*b* es mejor. Siempre se debe procurar encontrar la agrupación óptima.

Figura 14

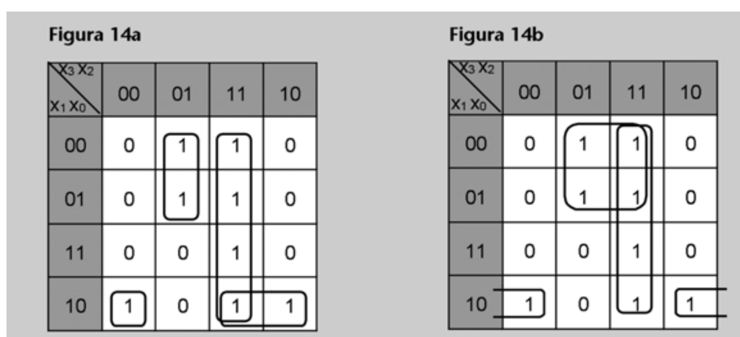


**Figura 14a** **Figura 14b**

### 3) Deducción de los términos producto

Tomamos el mapa de la figura 14*b*. El grupo de los cuatro unos de la tercera columna corresponde a las combinaciones $[x_3\ x_2\ x_1\ x_0]$ =

$$[1\ 1\ 0\ 0]$$
$$[1\ 1\ 0\ 1]$$
$$[1\ 1\ 1\ 0]$$
$$[1\ 1\ 1\ 1]$$

La expresión en suma de mintérminos que se obtendría de estos cuatro unos es $x_3x_2x_1'x_0' + x_3x_2x_1'x_0 + x_3x_2x_1x_0' + x_3x_2x_1x_0$. Si sacamos factor común, obtenemos $x_3x_2$, porque el valor de la función es independiente de $x_1$ y $x_0$. Si observamos el mapa, podemos ver que las variables $x_3$ y $x_2$ no cambian de valor en el rectángulo, mientras que $x_1$ y $x_0$ adoptan todas las combinaciones posibles.

Así pues, obtendremos un término producto de cada grupo de la siguiente manera:

**1.** Sólo aparecen las variables cuyo valor es constante para todas las casillas que forman el grupo.

**2.** Si en todas las casillas del grupo una variable vale 1, la variable aparece en el término *producto sin negar*.

**3.** Si en todas las casillas del grupo una variable vale 0, ésta aparece negada en el término *producto*.

Así, en el mapa de la figura 14*b*, del otro grupo de cuatro unos obtenemos el término

$$x_2x_1'.$$

Del grupo de dos unos obtenemos el término

$$x_2'x_1x_0'.$$

**Observad**

A partir de un rectángulo de $2^m$ unos obtenemos un término producto con $n - m$ variables, siendo $n$ el número de variables de la función. Por ejemplo, en el término $x_2'x_1x_0'$, que se obtiene a partir de un rectángulo de dos unos ($m = 1$, $n - m = 3$), hay tres variables, mientras que el término $x_2x_1$ tiene sólo dos variables, porque se obtiene de un rectángulo con cuatro unos ($m = 2$, $n - m = 2$).

### 4) Obtención de la expresión mínima de la función

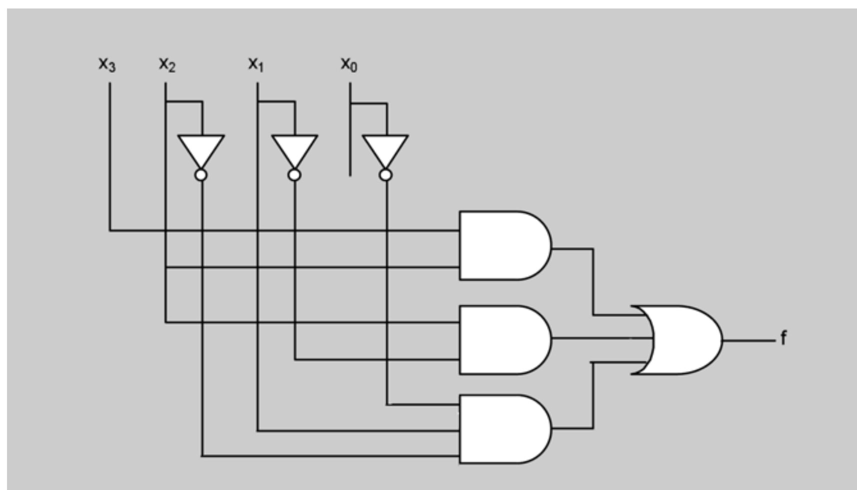
Ya sabemos que una función se puede expresar haciendo la suma lógica de todos sus mintérminos. Los términos *producto* obtenidos en el paso anterior "resumen" los mintérminos de una función. Por tanto, podemos expresar la función realizando la suma lógica de estos términos *producto*.

En el ejemplo de la figura 14*b*, la expresión minimizada de la función es la siguiente:

$$f(x_3,x_2,x_1,x_0) = x_3x_2 + x_2x_1' + x_2'x_1x_0'.$$

La figura 15 muestra el circuito a dos niveles que implementa esta función a partir de la expresión minimizada. Se puede ver que es menor y más barato que el que obtendríamos a partir de la expresión en suma de mintérminos original, ya que éste tendría cuatro puertas NOT, ocho AND de tres entradas cada una y una OR de ocho entradas. De todos los circuitos a dos niveles que implementan esta función, el de la figura 15 es el de dimensiones más reducidas y el más barato; se dice que es el **circuito mínimo a dos niveles**.

Figura 15




**Actividades**

**26.** Sintetizad mínimamente a dos niveles la función descrita en la actividad 7.

**27.** Sintetizad mínimamente a dos niveles la función descrita en la actividad 23*a*. Comparad la respuesta con la que habéis obtenido en la actividad 23*a*.

### 2.3.3. Minimización de funciones especificadas incompletamente

Tal como hemos visto en el apartado 1.5, el valor de las funciones especificadas incompletamente es indiferente para algunas combinaciones de las variables (las combinaciones "no importa"). Es decir, por las combinaciones "no importa" tanto podemos suponer que la función vale 1 como que vale 0.

En la tabla de verdad de la función ponemos x en las filas correspondientes a las combinaciones "no importa". En el mapa de Karnaugh también pondremos *x* en las casillas correspondientes. Dado que el valor de la función en estos casos es indiferente, supondremos que las *x* son un 1 o un 0, según lo que nos convenga más, teniendo en cuenta que siempre debemos procurar obtener el menor número posible de agrupaciones y que las agrupaciones sean tan grandes como sea posible.

Tomamos de ejemplo la función que hemos visto en el apartado 1.5. La figura 16 muestra su tabla de verdad y el mapa de Karnaugh, con las agrupaciones de casillas.

Figura 16

| $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | x |
| 0 | 1 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | x |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

| $x_0$ \ $x_2 x_1$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | x | 0 | 1 |
| 1 | x | x | 1 | x |

Nos interesa suponer que las *x* de la fila inferior valen 1, ya que así obtenemos un grupo de cuatro unos y un grupo de dos unos. Por tanto, la expresión mínima de la función es la siguiente:

$$f = x_0 + x_2 x_1'.$$

Fijémonos en que hemos tomado la *x* de la casilla $[x_2\ x_1\ x_0] = [0\ 1\ 0]$ como 0, ya que si no lo hiciéramos así obtendríamos un grupo adicional innecesariamente (los grupos deben cubrir todos los unos, pero no es necesario que cubran todas las *x*).

### Actividades

**28.** Sintetizad mínimamente a dos niveles la función descrita en la actividad 19.

# 3. Bloques combinacionales

Un **bloque combinacional** es un circuito lógico combinacional con una funcionalidad determinada. Está construido a partir de puertas, como los circuitos que hemos visto hasta ahora.

Hasta este momento, las "piezas" que hemos utilizado para sintetizar circuitos han sido puertas lógicas. Después de estudiar este capítulo, podremos utilizar también los bloques combinacionales como piezas para diseñar circuitos más complejos, como por ejemplo un computador, que no se pueden pensar a nivel de puertas pero sí a nivel de bloques.

**Existen muchos bloques combinacionales, en este curso veremos sólo los más básicos.**

## 3.1. Multiplexor. Multiplexor de buses. Demultiplexor

Imaginemos que en una ciudad hay tres calles que confluyen en otra calle de un solo carril. Será necesario un urbano o algún tipo de señalización para controlar que en cada momento circulen hacia la calle de salida los coches provenientes de una única calle confluente.

Un multiplexor es un bloque que cumple la función de guardia urbano en circuitos electrónicos. Tiene un determinado número de señales de entrada que "compiten" para conectarse a una señal única de salida, y unas señales de control que sirven para determinar qué señal de entrada se conecta en cada momento con la salida.

Más concretamente, las entradas y salida de un multiplexor son las siguientes:

**1.** $2^m$ entradas de *datos*, identificadas por la letra *e* y numeradas desde 0 hasta $2^m - 1$. Diremos que la entrada de datos numerada con el 0 es la de *menos peso* y, la numerada con el $2^m - 1$, la de *más peso*.

**2.** Una salida de datos, *s*.

**3.** *m* entradas *de control* o *de selección*, identificadas por la letra *c* y numeradas desde 0 hasta $m - 1$. Diremos que la entrada de control numerada con el 0 es la *de menos peso*, y la numerada con $m - 1$, la *de más peso*.

Para especificar el tamaño de un multiplexor, diremos que es un multiplexor $2^m$-1; por ejemplo, un multiplexor 4-1 es un multiplexor de cuatro entradas de datos y dos de control.

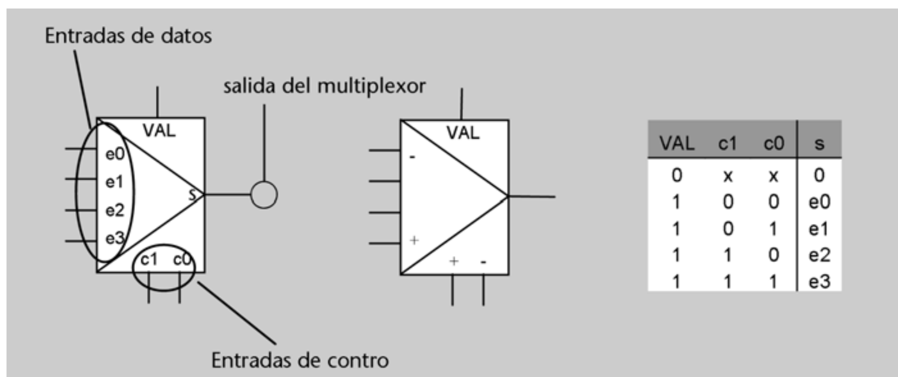
**4.** Una entrada de *validación*, que denominamos VAL.

La figura 17 muestra dos posibles representaciones gráficas de un multiplexor 4-1, es decir, con $m = 2$ (podéis ver el primer recuadro al margen). La diferencia entre éstas es que en la primera ponemos los nombres de las entradas, mientras que en la segunda sólo indicamos con los signos "+" y "–" cuáles son las de más y las de menos peso (se asume que el resto está ordenado en orden de peso). Las dos representaciones son igualmente válidas.

**Algunos de los bloques combinacionales que se describen en este capítulo, además de las puertas lógicas descritas en el capítulo 2, pueden encontrarse en el mercado en forma de chips. Cada una de las entradas o salidas de los circuitos se corresponden con una "pata" (llamada *pin*) del chip. Entonces, una puerta AND de dos entradas necesita 3 pins de un chip, 2 para sus entradas y 1 para la salida. Por otro lado, un multiplexor de 4-1 necesita 7 pins: 4 para las entradas de datos, 2 para las entradas de control y 1 para la salida de datos. Además, como las puertas y los bloques son circuitos electrónicos, necesitan conectarse a una fuente de alimentación (los circuitos lógicos funcionan con corriente continua). Por este motivo, todos los chips tienen 2 pins adicionales, uno para conectarse al positivo de la fuente de alimentación y otro para conectarse a masa (0 voltios).**

En general, no escribiremos el nombre de la salida (tal y como se hace en la segunda representación) porque la podemos identificar de manera inequívoca por el hecho de que está dibujada en el punto del multiplexor donde se juntan las dos líneas oblicuas interiores. También, en el caso de los multiplexores 2-1, que sólo tienen una entrada de control, podemos identificarla por la letra "c" (sin número) o bien no ponerle el nombre, ya que la podemos identificar inequívocamente por el hecho de que estará dibujada en un lado corto del multiplexor.

Figura 17




| VAL | c1 | c0 | s |
|---|---|---|---|
| 0 | x | x | 0 |
| 1 | 0 | 0 | e0 |
| 1 | 0 | 1 | e1 |
| 1 | 1 | 0 | e2 |
| 1 | 1 | 1 | e3 |

**Nota**

En esta figura la entrada de datos de más peso del multiplexor se encuentra en la parte inferior y la de menos peso, en la parte superior. Podemos invertir el orden si es más conveniente para la claridad de un circuito. Igualmente, las entradas de control pueden girarse (más peso en la derecha, menos en la izquierda).

La figura 17 también muestra la tabla de verdad que describe el funcionamiento del multiplexor (en este caso un multiplexor 4-1), que es el siguiente:

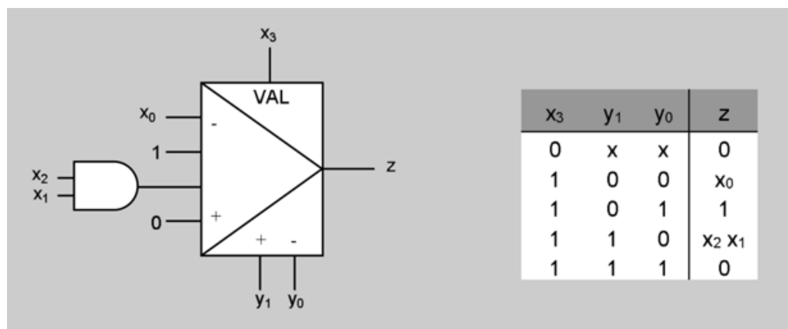
- Cuando la entrada de validación vale 0, la salida del multiplexor se pone en 0 independientemente del valor de las entradas de datos. En la tabla, esto lo reflejamos poniendo *x* en las columnas correspondientes a las entradas de datos, en la fila en la que VAL = 0.

- Cuando la entrada de validación está activa (vale 1), entonces las entradas de control determinan cuál de las entradas de datos se conecta con la salida de la forma siguiente: se interpretan las variables conectadas en las entradas de control ($c_1$ y $c_0$ en el ejemplo) como un número codificado en binario con $m$ bits (la entrada de más peso corresponde al bit de más peso); si el número codificado es $i$, la entrada de datos que se conecta con la salida es la numerada con el número $i$.

Si no dibujamos la entrada de validación en un multiplexor, asumiremos que ésta está en 1.

En las entradas y salida de un multiplexor conectaremos las señales lógicas que convengan para el funcionamiento de los circuitos. Por ejemplo, si conec-

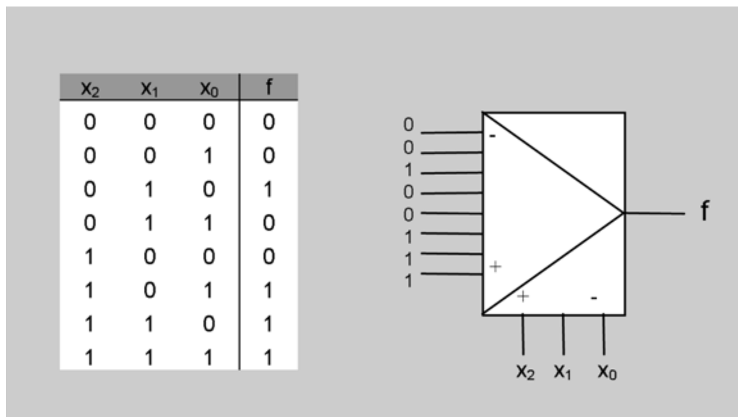
tamos las señales de entrada $x_3$, $x_2$, $x_1$, $x_0$, $y_1$, $y_0$ a un multiplexor 4-1 tal y como se muestra en la figura siguiente, la señal de salida $z$ tomará los valores que se describen en la tabla de verdad de la derecha.




| $x_3$ | $y_1$ | $y_0$ | z |
|---|---|---|---|
| 0 | x | x | 0 |
| 1 | 0 | 0 | $x_0$ |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | $x_2 x_1$ |
| 1 | 1 | 1 | 0 |

Una posible aplicación de los multiplexores es la implementación de funciones lógicas. La figura 18 muestra la implementación de una función con un multiplexor que tiene tantas entradas de control como variables tiene la función. La salida del multiplexor valdrá 1 si las variables conectadas a las señales de control construyen las combinaciones 2, 5, 6 ó 7, que corresponden a los casos en que la función $f$ vale 1.

Figura 18



| $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



**Multiplexor de buses**

Una **palabra de *n* bits** es una agrupación de *n* bits, usualmente con un significado semántico conjunto (por ejemplo, un número codificado en binario con *n* bits).

Por convención, usaremos las letras mayúsculas para dar nombre a una palabra o variable de *n* bits. Cada uno de los bits que la forman se identifica por la misma letra en minúscula, junto a un subíndice para identificar su peso. Por ejemplo,

$$A = a_7a_6a_5a_4a_3a_2a_1a_0 = a_{7..0}$$

Para referirnos a un subconjunto de los bits de una palabra escribiremos los subíndices correspondientes. Por ejemplo, $a_{4..1}$.

> Un **bus** es una agrupación de un cierto número de cables, por cada uno de los cuales circula un bit. Así, una palabra de *n* bits circulará por un bus de *n* bits. Diremos que es su *ancho* o *tamaño* del bus.

Los buses se representan gráficamente tal como se muestra en la figura 19*a*. En la línea que representa la señal ponemos una línea transversal acompañada de un número que indica la cantidad de bits del bus.

Al dibujar circuitos, es fundamental indicar cuántos bits son todos los buses que aparecen. Una línea sin especificación de número de bits corresponderá siempre a una señal de un solo bit.

A veces interesa disgregar los bits que forman un bus o agregarle bits adicionales. Las figuras 19*b*, 19*c* y 19*d* muestran unos cuantos ejemplos de cómo lo representaremos gráficamente.

Figura 19

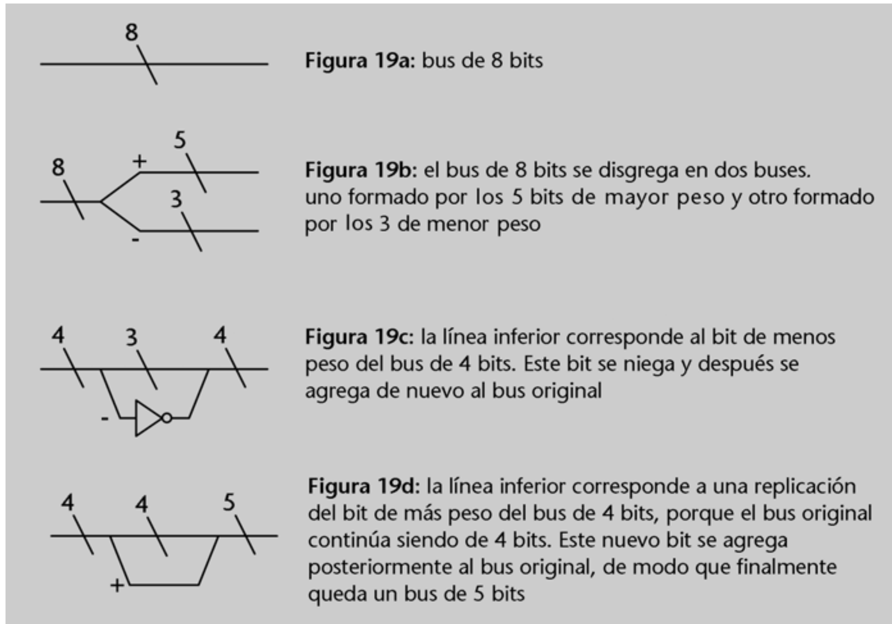


**Figura 19a:** bus de 8 bits

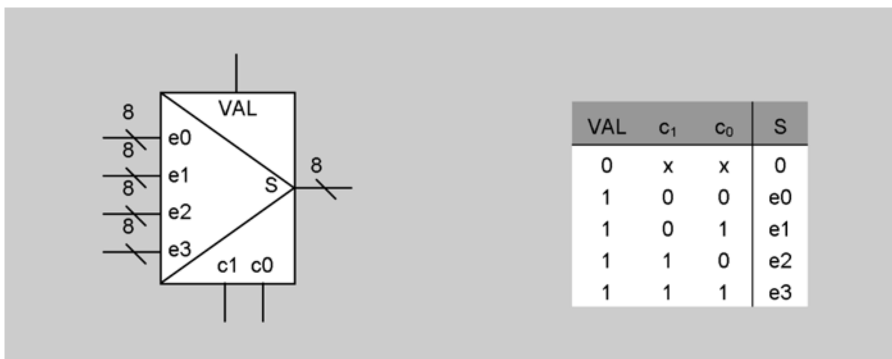
**Figura 19b:** el bus de 8 bits se disgrega en dos buses. uno formado por los 5 bits de mayor peso y otro formado por los 3 de menor peso

**Figura 19c:** la línea inferior corresponde al bit de menos peso del bus de 4 bits. Este bit se niega y después se agrega de nuevo al bus original

**Figura 19d:** la línea inferior corresponde a una replicación del bit de más peso del bus de 4 bits, porque el bus original continúa siendo de 4 bits. Este nuevo bit se agrega posteriormente al bus original, de modo que finalmente queda un bus de 5 bits

Un **multiplexor de buses** funciona igual que un multiplexor de bits, pero en este caso las entradas de datos y la salida son buses de un determinado número de bits. La figura 20 muestra la representación gráfica de un multiplexor 4-1 de buses de ocho bits.

Figura 20




| VAL | $c_1$ | $c_0$ | S |
|---|---|---|---|
| 0 | x | x | 0 |
| 1 | 0 | 0 | e0 |
| 1 | 0 | 1 | e1 |
| 1 | 1 | 0 | e2 |
| 1 | 1 | 1 | e3 |

### Actividades

**29.** Obtened la expresión algebraica de la salida de un multiplexor 4-1. Haced su implementación interna mediante puertas lógicas.

**30.** Implementad la función $f(a, b, c) = abc' + a'c$ con un multiplexor 8-1.

**31.** Implementad un circuito combinacional que permita multiplicar dos números enteros de dos bits representados en complemento a 2 utilizando sólo un multiplexor.

**32.** Diseñad un multiplexor de 16-1 utilizando dos multiplexores 8-1 y las puertas lógicas que hagan falta.

**33.** Construid un circuito con una entrada $X$ de 8 bits por la que llega un número natural representado en binario y una salida $Z$ también de 8 bits, de manera que $Z = X$ si $X$ es par y $Z = 0$ si $X$ es impar.

### Demultiplexor

Un **demultiplexor** hace la función inversa de un multiplexor: dada una señal de entrada, determina con qué señal de salida se debe conectar.

Los multiplexores tienen las señales siguientes:

Para especificar el tamaño de un demultiplexor, diremos que es un demultiplexor $1–2^m$; por ejemplo, un demultiplexor 1-4 tiene cuatro salidas de datos y dos entradas de control.

**1.** Una entrada de datos de $n$ bits, e ($n$ puede ser igual a 1).

**2.** $2^m$ salidas de datos de $n$ bits, identificadas por la letra $s$ y numeradas desde 0 hasta $2^m – 1$. Diremos que la salida de datos numerada con el 0 es la de menos peso, y la numerada con el $2^m – 1$, la de más peso.

**3.** $m$ entradas de control o de selección, de un bit, identificadas por la letra $c$ y numeradas desde 0 hasta $m – 1$. Diremos que la entrada de control numerada con el 0 es la de menos peso, y la numerada con $m – 1$, la de más peso.

**4.** Una entrada de validación de un bit, que denominamos VAL.

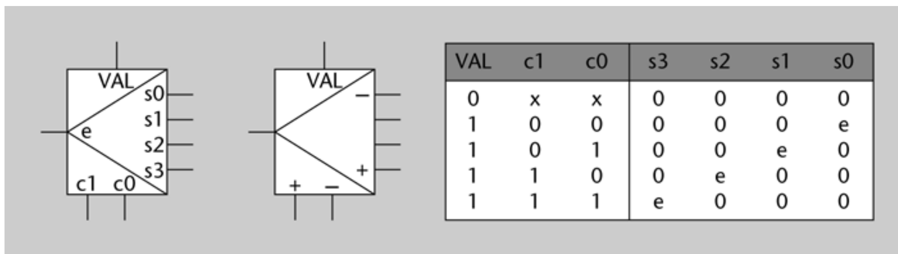
El funcionamiento del demultiplexor es el siguiente:

- Cuando la entrada de validación vale 0, todas las salidas valen 0.

- Cuando la entrada de validación vale 1, entonces las entradas de control determinan cuál de las salidas de datos se conecta con la entrada, de la misma manera que en el caso del multiplexor (si el número que codifican las entradas de control es $i$, la entrada se conecta con la salida numerada con el número $i$). Las salidas de datos no seleccionados se ponen a 0.

La figura siguiente muestra las dos posibilidades para la representación gráfica de un demultiplexor 1-4 y la tabla que describe su funcionamiento. Al igual que en el caso de los multiplexores, podemos usar los signos "+" y "–" para indicar el peso de las entradas de control y de las salidas de datos. También, al

igual que en el caso de los multiplexores, en general no escribiremos el nombre de la entrada *e*, y si no dibujamos la entrada VAL, asumiremos que está activa.




| VAL | c1 | c0 | s3 | s2 | s1 | s0 |
|---|---|---|---|---|---|---|
| 0 | x | x | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | e |
| 1 | 0 | 1 | 0 | 0 | e | 0 |
| 1 | 1 | 0 | 0 | e | 0 | 0 |
| 1 | 1 | 1 | e | 0 | 0 | 0 |

**Actividades**

**34.** Se quiere diseñar un circuito que controle el acceso a una sala de conciertos que tiene en la entrada dos luces, una verde y otra roja, de manera que en cada momento sólo está encendida una de las dos (la roja si la sala está llena, la verde si todavía cabe gente). El circuito recibe como entrada una señal *lleno* que vale 1 cuando la sala está llena y 0 cuando todavía no. Para encender la luz verde, se debe activar la señal *verd*e, y para encender la roja se tiene que activar la señal *rojo*. Implementad el circuito usando sólo un demultiplexor.

### 3.2. Codificadores y descodificadores

La función de un **codificador** es generar la codificación binaria de un número.

Los codificadores tienen las siguientes señales:

**1.** Una entrada de validación, VAL, que funciona igual que en el caso de los multiplexores: si vale 0, todas las salidas valen 0 (cuando no dibujemos la entrada de validación, asumiremos que vale 1).

**2.** $2^m$ entradas de datos (de un bit), identificadas por la letra *e* y numeradas de 0 a $2^m - 1$ (la de número más alto es la de más peso).

**3.** *m* salidas de datos (de un bit), identificadas por la letra *s* y numeradas de 0 a $m - 1$, que se interpretan como si formaran un número codificado en binario con *m* bits (la salida de más peso corresponde al bit de mayor peso).

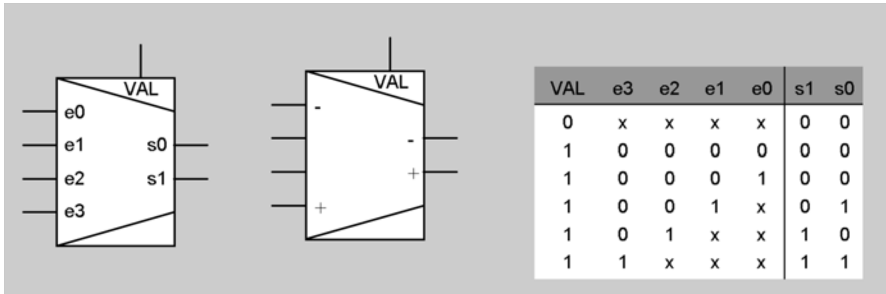
El funcionamiento de un codificador es el siguiente:

Cuando la entrada de validación vale 1, si la entrada de mayor peso de entre las que están en 1 es la numerada con el número *i*, entonces las salidas codifican en binario el número *i*. O, dicho de otro modo: para que un codificador genere la representación binaria del número *i*, es preciso que la entrada activa de más peso sea la numerada con el número *i*.

La figura 21 muestra la representación gráfica de un codificador 4-2 y la tabla de verdad que explica su funcionamiento (observamos que, como en el caso de los multiplexores, podemos utilizar los símbolos "+" y "–" en lugar de los nombres de las entradas y salidas). Vemos que, por ejemplo, cuando $e3 = 0$ y $e2 = 1$, las salidas codifican el número 2, independientemente del valor de $e1$ y $e0$, porque la entrada de más peso que está activa es $e2$. Se dice que, cuanto más peso tiene una entrada, más prioridad tiene.

Para especificar el tamaño de un codificador, diremos que es un codificador $2^m$ – $m$; por ejemplo, un codificador 4-2 es un codificador de cuatro entradas y dos salidas.

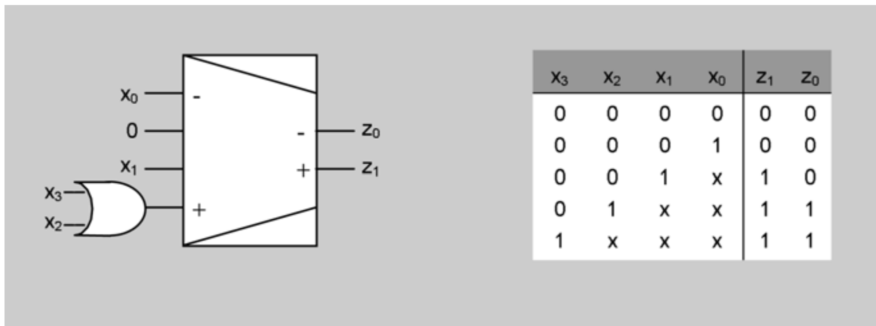
Figura 21




| VAL | e3 | e2 | e1 | e0 | s1 | s0 |
|---|---|---|---|---|---|---|
| 0 | x | x | x | x | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | x | 0 | 1 |
| 1 | 0 | 1 | x | x | 1 | 0 |
| 1 | 1 | x | x | x | 1 | 1 |

Fijémonos en que las salidas del codificador valen 0 tanto si no hay ninguna entrada en 1 como si se encuentra en 1 la entrada de menos peso (y también cuando la entrada VAL es 0).

La siguiente figura muestra un ejemplo de uso de un codificador y la tabla de verdad que describe el funcionamiento del circuito.




| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $z_1$ | $z_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | x | 1 | 0 |
| 0 | 1 | x | x | 1 | 1 |
| 1 | x | x | x | 1 | 1 |

**Recordemos que si en un codificador no dibujamos la entrada de validación, asumiremos que ésta se encuentra en 1.**

Los **descodificadores** cumplen la función inversa a los codificadores: dada una combinación binaria presente en la entrada, indican a qué número decimal corresponde.

Los descodificadores tienen las siguientes señales:

**1.** Una entrada de validación.

**2.** $m$ entradas de datos, identificadas por la letra $e$ y numeradas de 0 a $m$ – 1, que se interpretan como si formaran un número codificado en binario (la entrada de más peso corresponde al bit de mayor peso).

**3.** $2^m$ salidas, identificadas por la letra $s$ y numeradas de 0 a $2^m$ – 1, de las cuales sólo una vale 1 en cada momento (si la entrada de validación está en 0, entonces todas las salidas valen 0).

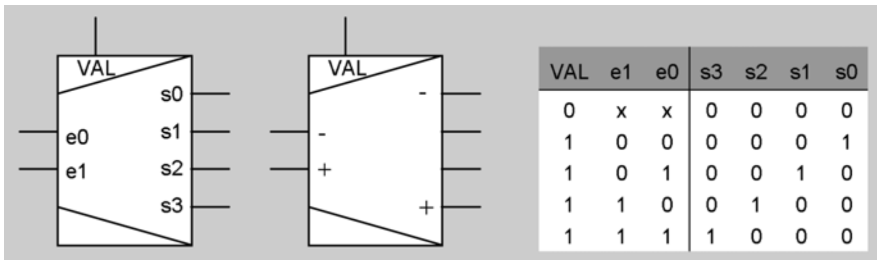
El funcionamiento de un descodificador es el siguiente:

> Cuando la entrada de validación vale 1, si las entradas codifican en binario el número *i*, entonces se pone en 1 la salida numerada como *i*.

**Para identificar el tamaño de los descodificadores utilizaremos la misma convención que en el caso de los codificadores; por ejemplo, descodificador 3-8.**

La figura 22 muestra la representación gráfica de un descodificador 2-4 y la tabla de verdad que describe su funcionamiento.

Figura 22



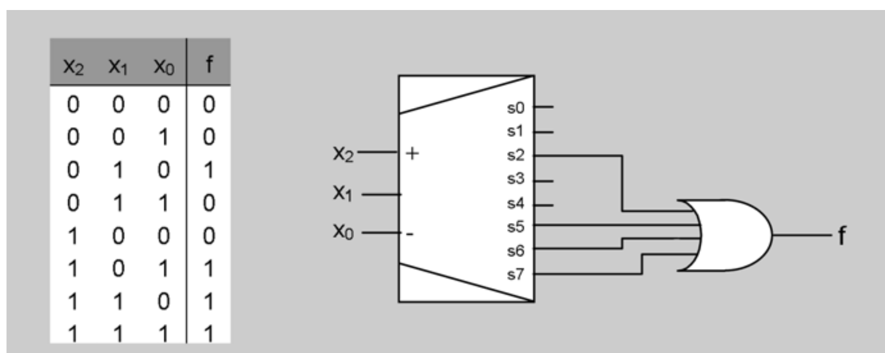

| VAL | e1 | e0 | s3 | s2 | s1 | s0 |
|---|---|---|---|---|---|---|
| 0 | x | x | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Los descodificadores también se pueden utilizar para implementar funciones lógicas. Si la función tiene $n$ variables, usaremos un descodificador $n - 2^n$ y conectaremos las variables en las entradas en orden de peso. De este modo, la salida $i$ del descodificador se pondrá en 1 cuando las variables, interpretadas como bits de un número binario, codifiquen el número $i$. Por ejemplo, si conectamos las variables $[x_1\ x_0]$ a las entradas $[e_1\ e_0]$ del descodificador de la figura 22, la salida $s2$ valdrá 1 cuando $[x_1\ x_0] = [1\ 0]$.

Por tanto, para implementar la función será suficiente con conectar las salidas correspondientes a las combinaciones que hacen que la función valga 1 en una puerta OR. Cuando alguna de estas combinaciones se encuentre presente en la entrada del descodificador, la salida correspondiente se pondrá en 1 y, por tanto, de la puerta OR saldrá un 1. Cuando las variables construyan una combinación que haga que la función valga 0, todas las entradas de la puerta OR valdrán 0 y, por tanto, también su salida.

La figura 23 muestra un ejemplo de esto. Podemos comprobar que la salida de la puerta OR valdrá 1 cuando valga 1 la salida $s2$ del descodificador, o la $s5$, la $s6$ o la $s7$. Es decir, cuando las variables de entrada construyan alguna de las combinaciones que hagan que la función valga 1.

Figura 23




| $x_2$ | $x_1$ | $x_0$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Si no dibujamos la entrada de validación en un descodificador, asumiremos que está en 1.**

Cuando diseñamos un circuito usando bloques, podemos dejar una o más salidas de los bloques sin conectar a ningún sitio. Sin embargo, no podemos dejar ninguna entrada sin conectar, puesto que el comportamiento sería indeterminado.

### Actividades

**35.** Deducid la implementación interna (mediante puertas lógicas) de un codificador 4-2.

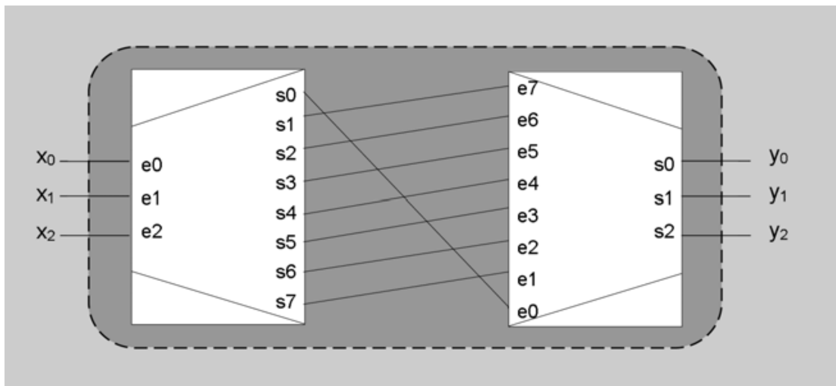
**36.** Deducid la implementación interna (mediante puertas lógicas) de un descodificador 2-4.

**37.** Implementad la función $f(a, b, c) = abc' + a'c$ con un descodificador 3-8.

**38.** Diseñad un circuito que genere la representación en signo y magnitud de números en el rango [–7, 7]. El circuito debe tener estas entradas y salidas:

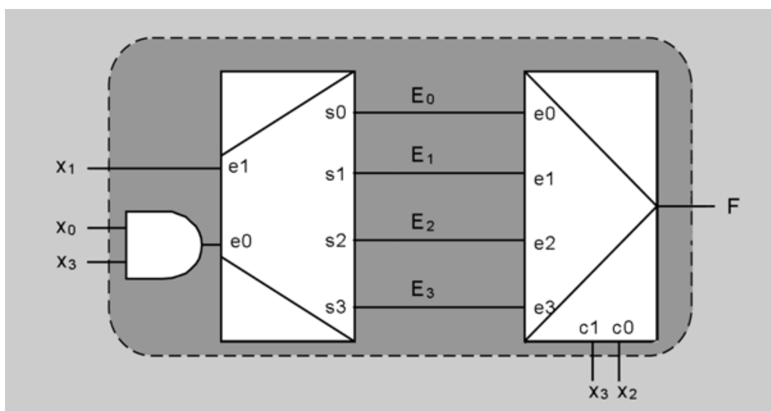
- Ocho entradas de un bit, *e*7, *e*6, ... *e*0, de las cuales como mucho una estará activa en cada momento. Si la que está a 1 es *ei*, la magnitud del número que hay que representar es *i*.
- Otra entrada de un bit *sg*, que indica el signo del número que hay que representar (0 positivo, 1 negativo).
- Cuatro salidas de un bit, *s*3... *s*0, que contendrán la representación en signo y magnitud del número que se quiere representar. El número 0 se representará siempre con el bit de signo a 0.
- Otra salida de un bit, *null*, que valdrá 1 sólo cuando no se indique ninguna magnitud para ser representada. En este caso, las salidas *s*3... *s*0 también valdrán 0.

**39.** ¿Qué hace el siguiente circuito suponiendo que $X = (x_2, x_1, x_0)$ y $Y = (y_2, y_1, y_0)$ son números enteros codificados en complemento a 2?




**40.** Construid un circuito que implemente la función $Y = (X + 3) \bmod 4$, siendo $X$ e $Y$ números naturales representados en binario con 2 bits. El circuito sólo puede contener dos bloques y ninguna puerta.

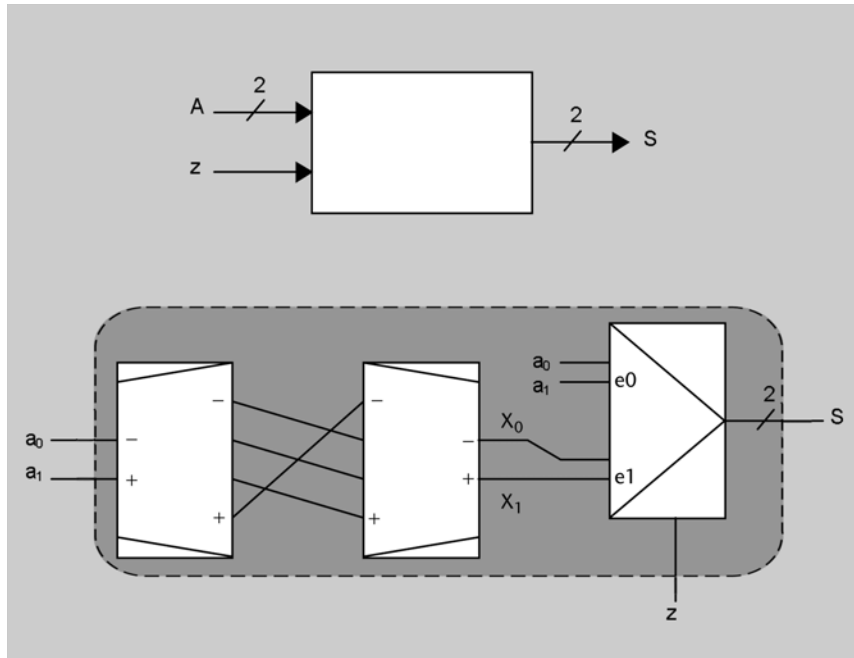
**41.** Dado el siguiente circuito:



**a)** Encontrad las expresiones algebraicas de $E_i$ ($i$ = 0, 1, 2, 3) en función de $x_3$, $x_1$ y $x_0$.
**b)** Escribid la tabla de verdad de $F(x_3, x_2, x_1, x_0)$.

**42.** Diseñad un descodificador 3-8 utilizando dos descodificadores 2-4 y las puertas lógicas que hagan falta.

**43.** ¿Qué función cumple el siguiente circuito si interpretamos las entradas y la salida como números codificados en binario natural?




### 3.3. Desplazadores lógicos y aritméticos

Un **desplazador** es un bloque combinacional que tiene la función de desplazar bits hacia la izquierda o hacia la derecha.

Los desplazadores tienen una señal de entrada y una señal de salida, las dos de $n$ bits. La señal de salida se obtiene desplazando los bits de entrada $m$ veces hacia la derecha o hacia la izquierda.

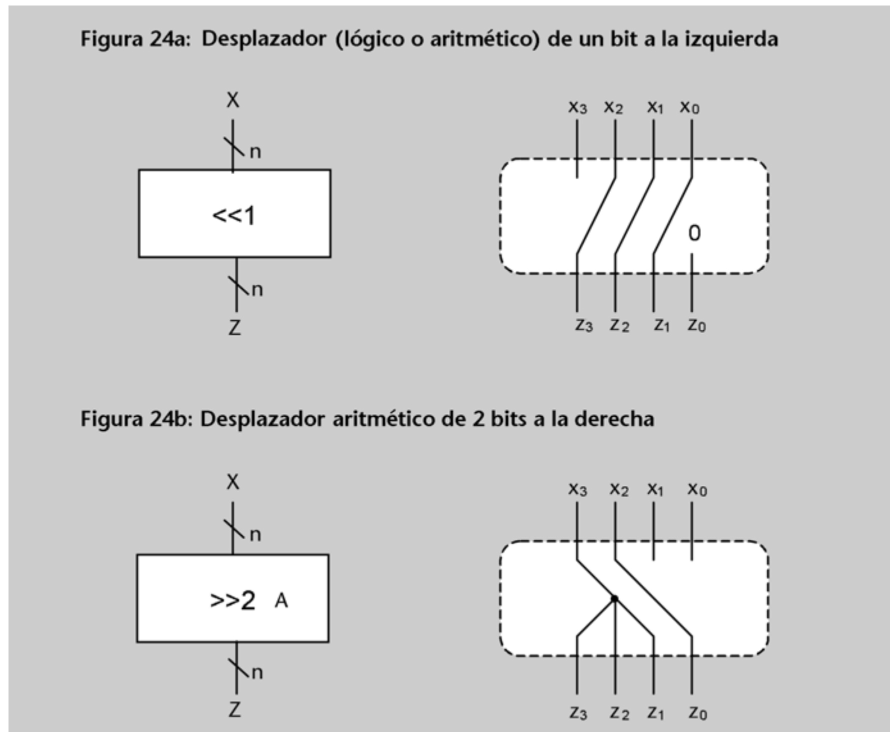
**1.** Si el desplazamiento es hacia la izquierda, los $m$ bits de menos peso de la salida se ponen a 0.

**2.** Si el desplazamiento es hacia la derecha, existen dos posibilidades para los $m$ bits de más peso de la salida:

– En los **desplazadores lógicos** se ponen a 0.
– En los **desplazadores aritméticos** toman el valor del bit de más peso de la entrada. Fijémonos en que, si interpretamos las entradas y salidas como números codificados en complemento a 2 con $n$ bits, los desplazadores aritméticos mantienen en la salida el signo de la entrada.

Las figuras 24*a* y 24*b* muestran la representación gráfica y la implementación interna de un desplazador de un bit a la izquierda y de un desplazador aritmético de dos bits a la derecha, respectivamente, de señales de cuatro bits. En el caso de los desplazadores a la derecha, usaremos la letra *L* para indicar que son lógicos y la letra *A* para indicar que son aritméticos.

Figura 24




Si interpretamos las entradas y salidas como codificaciones de números naturales, podemos decir que los desplazadores cumplen las funciones de multiplicar y dividir por potencias de 2 (división entera). Un desplazador de *m* bits a la izquierda multiplica por $2^m$, y un desplazador lógico de *m* bits a la derecha realiza la división entera por $2^m$. Si interpretamos los números como codificados en complemento a 2, entonces para dividir por $2^m$ hay que usar desplazadores aritméticos.

### Actividades

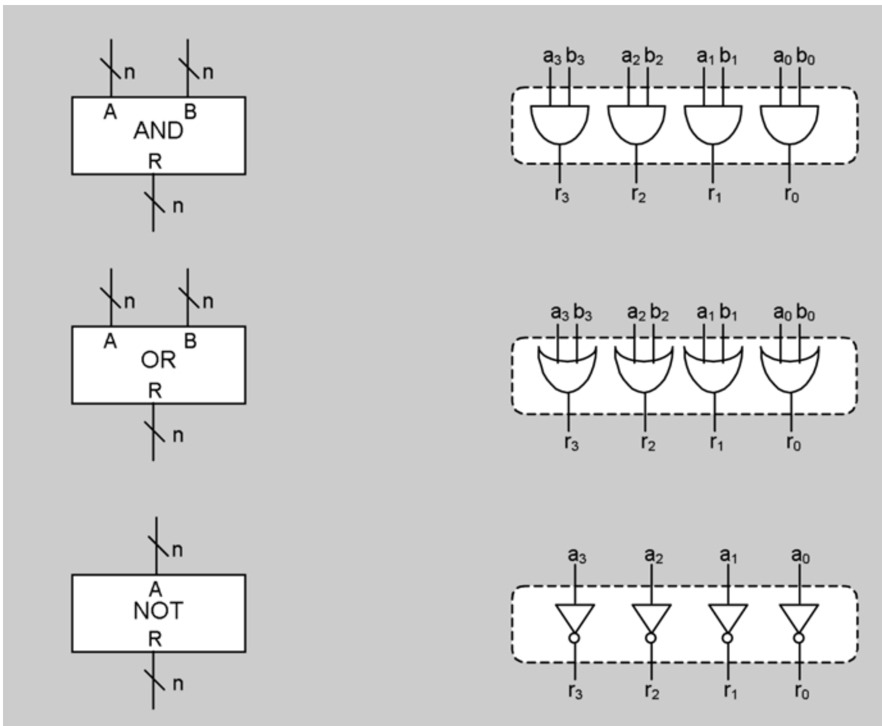
**44.** Contestad los siguientes apartados:
**a)** Implementad un circuito que pueda actuar como un desplazador de un bit a la izquierda de señales de cuatro bits usando un multiplexor 2-1 de buses de cuatro bits. Una señal de control de un bit, *d*, determina si el número de entrada se debe desplazar o no.
**b)** ¿A qué operación aritmética equivale este desplazamiento?
**c)** Indicad cuándo se produciría desbordamiento en los casos de interpretar la señal de entrada como un número natural codificado en binario o bien como un número entero codificado en complemento a 2.

## 3.4. Bloques AND, OR y NOT

Estos bloques hacen las operaciones AND, OR y NOT bit a bit sobre entradas de *n* bits.

Tienen dos entradas de datos (sólo una en el caso del bloque NOT) y una salida, todas de $n$ bits. La figura 25 muestra su representación gráfica y la implementación interna para el caso $n = 4$.

Figura 25



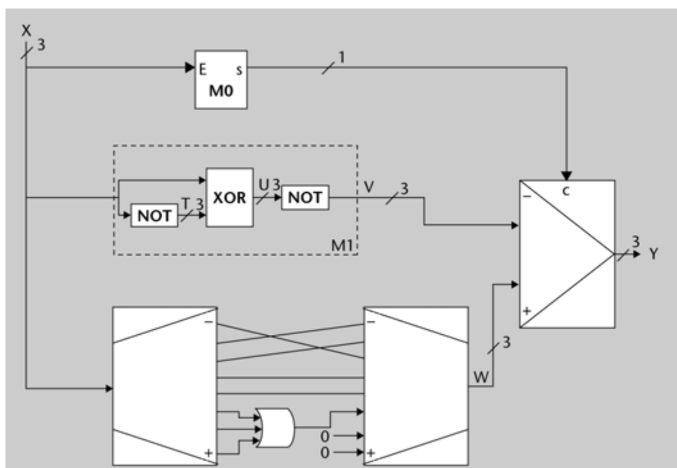

Podemos construir bloques análogos a estos, por ejemplo un bloque XOR sería aquel que hace la XOR bit a bit de las entradas.

### Actividades

**45.** Diseñad un circuito con una entrada $X$ por la que llegan números naturales representados en binario con 8 bits y una salida $Z$ de 8 bits, de manera que $Z = X'$ si $X$ es múltiple de 4, y $Z = x_3x_200$ si no lo es.

**46.** Sea el circuito lógico combinacional siguiente, en el que la entrada $X$ y la salida $Y$ codifican números naturales en binario y 3 bits y el bloque $M0$ tiene este comportamiento:

$$s = x_2x_1x_0 + x_2x_1x_0' + x_2x_1'x_0' + x_2x_1'x_0 + x_2'x_1x_0$$



**a)**Implementad el bloque *M*0 con el mínimo número de bloques combinacionales y, si fuera necesario, puertas lógicas.

**b)**¿Qué función hace el subcircuito identificado como *M*1?

**c)**Escribid la tabla de verdad del circuito completo.

### 3.5. Memoria ROM

La **memoria ROM** es un bloque combinacional que permite guardar el valor de $2^m$ palabras de $n$ bits.

**ROM**

La denominación ROM deriva del inglés *read only memory*, porque se refiere a las memorias en las que no se pueden hacer escrituras, sino sólo lecturas.

Podemos ver una memoria ROM como un archivador con cajones que guardan bits. Cada cajón tiene capacidad para guardar un cierto número de bits (todos tienen la misma) y el archivador tiene un número determinado de cajones, que es siempre una potencia de dos.

La memoria ROM tiene los siguientes elementos:

**1.** $2^m$ palabras o datos de $n$ bits, cada una en una posición (cajón) diferente de la memoria ROM. Las posiciones que contienen los datos están numeradas desde el 0 hasta el $2^m - 1$; estos números se llaman **direcciones**.

**2.** Una entrada de direcciones de $m$ bits, que se identifica con símbolo @. Los $m$ bits de la entrada de direcciones se interpretan como números codificados en binario (y, por tanto, es necesario determinar el peso de cada bit).

**3.** Una salida de datos de $n$ bits.

El funcionamiento de la ROM es el siguiente:

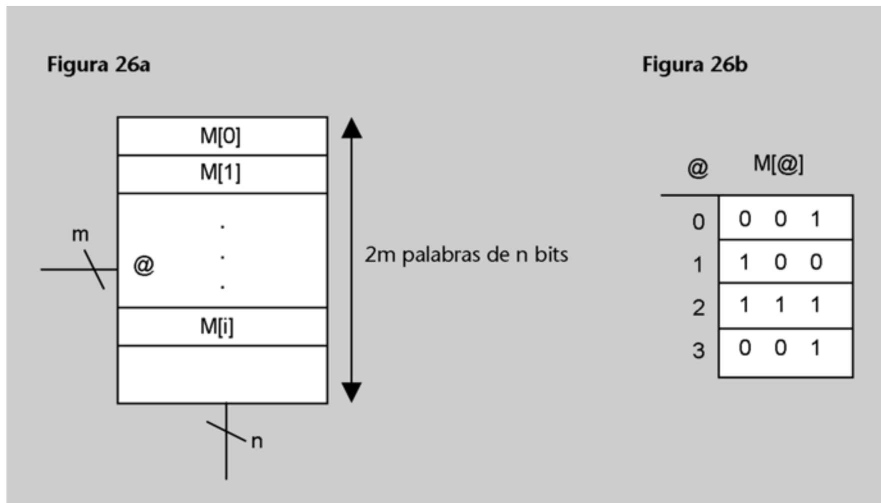
Cuando los $m$ bits de la entrada de direcciones (interpretados en binario) codifican el número $i$, entonces la salida toma el valor del dato que hay almacenado en la dirección $i$. Para referirnos a este dato usaremos la notación **M[*i*]**, y diremos que **leemos** el dato de la dirección $i$.

Así pues, sólo se puede acceder al valor de una palabra (leerla) en cada instante (es como si en cada momento sólo se pudiera abrir un cajón).

La figura 26*a* muestra cómo representaremos la memoria ROM. La figura 26*b* muestra un posible contenido de una memoria ROM de cuatro palabras de 3 bits. En este ejemplo,

M[0] = 001
M[1] = 100
M[2] = 111
M[3] = 001.

Figura 26




La memoria ROM también se puede utilizar para sintetizar funciones lógicas. Por ejemplo, si conectamos las variables $x_1$ y $x_0$ (en orden de peso) a las entradas de direcciones de la ROM de la figura 26b, entonces podemos interpretar los tres bits de salida como la implementación de las tres funciones siguientes:

| $x_1$ | $x_0$ | $f_2$ | $f_1$ | $f_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

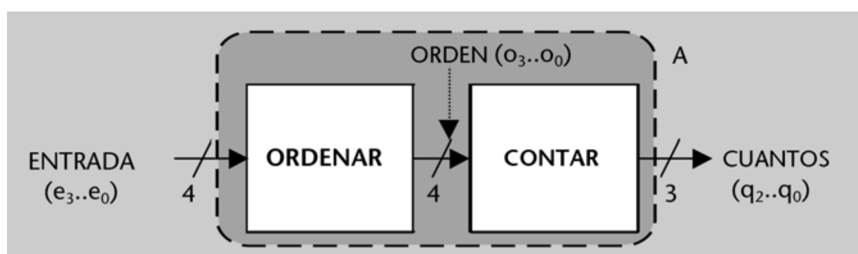
## Actividades

**47.** Indicad el tamaño y el contenido de una memoria ROM que implemente la función descrita en la actividad 24.

**48.** Indicad el tamaño y el contenido de una memoria ROM que implemente la función descrita en la actividad 19.

**49.** Utilizando sólo una memoria ROM, se quiere diseñar un circuito que, dado un número $X$ representado en signo y magnitud con 8 bits, dé a la salida el mismo número representado en complemento a 2 y 8 bits. Describid el contenido que debe tener la memoria ROM (sin escribir todo su contenido) e indicad el tamaño mínimo que debe tener. Escribid el contenido de las palabras de las direcciones 50, 100 y 150.

**50.** El circuito combinacional de la figura cuenta el número de unos que tiene una palabra de entrada de cuatro bits.



El bloque ORDENAR ordena la palabra ENTRADA y coloca todos los unos a la derecha y todos los ceros a la izquierda. Por ejemplo:

Si ENTRADA = 0101, entonces ORDEN = 0011.
Si ENTRADA = 0000, entonces ORDEN = 0000.

El bloque CONTAR genera en la salida CUANTOS la codificación binaria de la cantidad de unos de la palabra ORDEN. Por ejemplo:

Si ORDEN = 0011, entonces CUANTOS = 010 (2).
Si ORDEN = 0000, entonces CUANTOS = 000 (0).

Resolved los siguientes apartados:

**a)** Escribid la tabla de verdad del bloque ORDENAR.
**b)** Diseñad el bloque CONTAR utilizando sólo bloques combinacionales (exceptuando memoria ROM), del tamaño que haga falta.
**c)** Diseñad el circuito combinacional completo A con una memoria ROM e indicad su tamaño y contenido.

### 3.6. Comparador

El **comparador** es un bloque combinacional que compara dos números codificados en binario e indica qué relación existe entre éstos.

Un comparador tiene las siguientes señales:

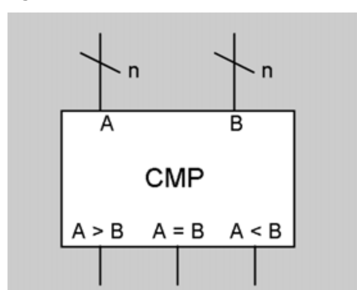
**1.** Dos entradas de datos de *n* bits, que reciben los nombres *A* y *B*. Se interpretan como números naturales codificados en binario.

**2.** Tres salidas de 1 bit, de las que sólo una vale 1 en cada momento:

– La salida $A > B$ vale 1 si el número que llega por la entrada *A* es mayor que el que llega por la entrada *B*.
– La salida $A = B$ vale 1 si los dos números de entrada son iguales.
– La salida $A < B$ vale 1 si el número que llega por la entrada *A* es menor que el que llega por la entrada *B*.

La figura 27 muestra la representación gráfica de un comparador.

Figura 27



### Actividades

**51.** Deducid las expresiones algebraicas de las funciones de salida $A = B$ y $A < B$ de un comparador de números de 2 bits y haced su implementación mediante puertas lógicas.

**52.** Diseñad un circuito que obtenga en su salida el máximo de dos números naturales, $X$ e $Y$, de 4 bits.

**53.** Utilizando cualesquiera de los bloques y puertas que se han visto, diseñad un circuito con dos entradas $X$ e $Y$ que codifican números naturales en binario y 3 bits y una salida $S$ de 2 bits, que funcione de la manera siguiente:

- Si $X > Y$, $S$ debe valer 01.
- Si $X < Y$, $S$ debe valer 10.
- Si $X = Y$ y son diferentes de 0, $S$ debe valer 00.
- Si $X = Y$ y son iguales a 0, $S$ debe valer 11.

**54.** Se dispone de dos hornos, cada uno de los cuales viene equipado con un termómetro digital para medir la temperatura interior. Los termómetros generan una señal de 8 bits que codifica la temperatura en binario (los hornos siempre están entre 0 °C y 255 °C).

Utilizando cualesquiera de los bloques y puertas que se han visto, diseñad un circuito lógico combinacional que, recibiendo por las entradas $A$ y $B$ la temperatura de los dos hornos, informe mediante la salida $R$, de 2 bits, en qué rango de temperaturas se encuentra el horno que está más caliente de los dos; si los dos están a la misma temperatura, indicad en qué rango se encuentra esta. La correspondencia entre rangos de temperaturas y valor de la señal $R$ es como sigue:

| Temperatura del horno que está más caliente (o igual) | *R* |
|---|---|
| [11000000, 11111111] | 11 |
| [10000000, 10111111] | 10 |
| [01000000, 01111111] | 01 |
| [00000000, 00111111] | 00 |

## 3.7. Sumador

El **sumador** es un bloque combinacional que realiza la suma de dos números codificados en binario o bien en complemento a 2.

**Recordad**

Tal como se ha estudiado en el módulo "Representación de la información numérica", la representación de los enteros en complemento a 2 permite hacer las sumas utilizando la misma mecánica que en las sumas de números codificados en binario. Por tanto, si un bloque realiza la suma de números codificados en binario, su salida también será la suma correcta si interpretamos los números de entrada como enteros codificados en complemento a 2. Por eso decimos que el sumador suma números codificados en binario o bien en complemento a 2.

La figura 28 muestra la representación gráfica de un sumador de $n$ bits. Las señales que tiene son las siguientes:

**1.** Dos entradas de datos de $n$ bits, que llamaremos $A$ y $B$, por donde llegarán los números que se tienen que sumar.

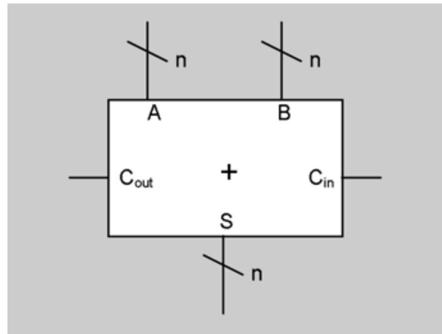
**2.** Una salida de $n$ bits, llamada $S$, que tomará el valor de la suma de los números $A$ y $B$.

**3.** Una salida de 1 bit, $C_{out}$, que valdrá 1 si cuando se haga la suma se produce acarreo en el bit de más peso.

La nomenclatura de las señales de entrada y salida de acarreo (*C*) deriva de *carry*, que es la palabra inglesa que se utiliza para acarreo.

**4.** Una entrada de 1 bit, $C_{in}$, por donde llega un acarreo de entrada.

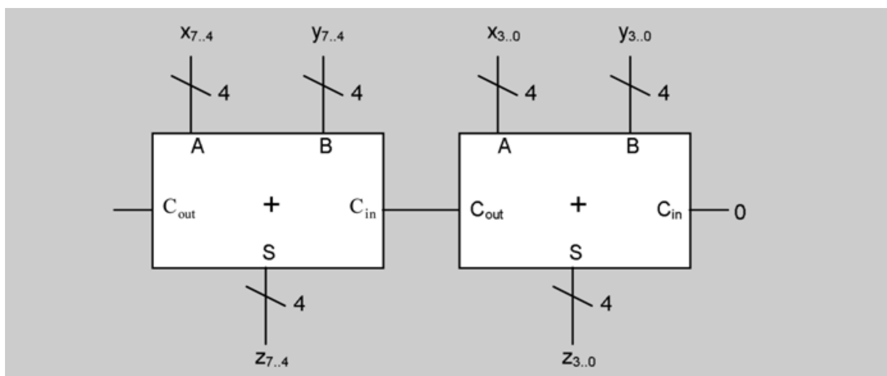
Figura 28




La entrada $C_{in}$ es útil cuando se quieren sumar números de $2 \cdot n$ bits y sólo se dispone de sumadores de $n$ bits. En este caso se encadenan dos sumadores: el primero suma los $n$ bits más bajos de los números y el segundo, los $n$ bits más altos. La salida de $C_{out}$ del primer sumador se conecta con la entrada $C_{in}$ del segundo para que el resultado sea correcto. La figura 29 ofrece un ejemplo para el caso $n = 4$, en el que hacemos la suma $Z = X + Y$, siendo $X$, $Y$ y $Z$ números de 8 bits. Fijémonos en que el sumador que suma los bits más bajos lo dibujamos a la derecha, porque así los bits quedan ordenados de la manera en que estamos acostumbrados a verlos.

Si no necesitamos tener en cuenta un acarreo de entrada, conectaremos un 0 a la entrada $C_{in}$.

Figura 29




Como ya sabemos, el hecho de limitar la longitud de los números a un determinado número de bits ($n$) tiene como consecuencia que el resultado de una suma no sea siempre correcto (es incorrecto cuando se produce exceso de capacidad, es decir, cuando el resultado requiere más de $n$ bits para ser codificado). En las sumas binarias, sabemos que si se produce acarreo en el bit de más peso entonces el resultado es incorrecto. En cambio, en las sumas en complemento a 2 no existe ninguna relación entre el acarreo y el desbordamiento.

**Recordad**

El concepto de desbordamiento que se ha estudiado en el módulo "Representación de la información numérica".

> Por tanto, la salida $C_{out}$ de un sumador nos indica que se ha producido un desbordamiento si interpretamos las entradas en binario, pero no nos dice nada sobre si el resultado es correcto si las interpretamos en complemento a 2.

Dado que $X - Y = X + (-Y)$, los sumadores se pueden utilizar también para hacer la resta de dos números, si cambiamos el signo del segundo operando antes de conectarlo al sumador.

**Recordad**

La operación de cambio de signo que se ha estudiado en el módulo "Representación de la información numérica".

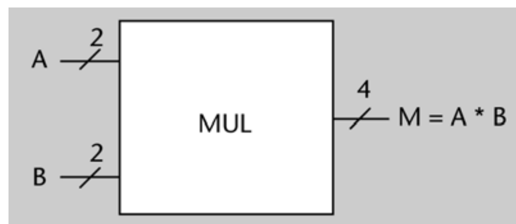
### Actividades

**55.** Suponed que X es un número natural codificado con 3 bits e implementad la función $Z = (3 \cdot X)$ usando sólo un sumador de cuatro bits y un desplazador.

**56.** Diseñad un circuito que haga la operación $Z = X - Y$ siendo $X$, $Y$ y $Z$ números codificados en complemento a 2 con n bits.

**57.** Usando bloques combinacionales, diseñad un circuito que dado un número entero $X$ codificado en complemento a 2 y 3 bits obtenga su valor absoluto, también en complemento a 2 y 3 bits. El circuito debe indicar si se produce desbordamiento en el cálculo poniendo la salida $V$ (de un bit) a 1.

**58.** Se dispone de un circuito combinacional que implementa un multiplicador de dos números enteros, representados en complemento a 2 con dos bits. La salida de un circuito multiplicador de números enteros siempre tiene el doble de bits que las entradas.




**a)** Escribid la tabla de verdad del bloque MUL.
**b)** Implementad con bloques combinacionales un circuito que calcule la siguiente operación:

$$Z = A^4 + A^3 + A^2,$$

donde $A$ es un número entero de dos bits representado en complemento a 2. Podéis usar bloques de cualquier número de bits (especificad de cuántos y justificad la respuesta), incluso el del apartado *a*, que no es necesario diseñar. Una posible solución sólo requiere un sumador y dos multiplicadores.

**59.** $A$, $B$, $C$ y $D$ son señales de 8 bits que codifican números naturales. Se quiere diseñar un circuito que implemente la función $Y$, que se describe así:

| | | |
|---|---|---|
| $Y = 2 \times C$ | si | $C = D$ |
| $Y = (A + B + C) / 4$ | si | $C \neq D$ |

Los buses de entrada y salida de cada bloque deben tener el menor ancho posible para conseguir que en ningún momento se produzcan desbordamientos.

## 3.8. Unidad aritmética y lógica (UAL)

Una **unidad aritmética y lógica** es un aparato capaz de efectuar un determinado conjunto de operaciones aritméticas y lógicas sobre dos números de entrada codificados en binario o en complemento a 2.

Las unidades aritméticas y lógicas se llaman *UAL* o, también, *ALU* (del inglés *arithmetic and logic unit*).

Para diseñar una UAL es necesario especificar el conjunto de operaciones que queremos que haga. Por ejemplo, una UAL puede hacer la suma, la resta, la AND y la OR de los operandos de entrada. En cada momento se especificará en la UAL cuál es la operación que debe hacer.

En este curso sólo estudiaremos UAL que operen con números naturales y enteros. Los procesadores tienen, además, UAL para operar con números reales codificados en coma flotante.

Las señales que tiene una UAL son las siguientes:

**1.** Dos entradas de datos de $n$ bits, $A$ y $B$, por donde llegarán los números sobre los que se deben hacer las operaciones.

**2.** Una salida de datos de $n$ bits, $R$, donde se obtendrá el resultado de la operación.

**3.** Un cierto número de entradas de control, $c_i$, de un bit cada una. Si la UAL es capaz de hacer $2^m$ operaciones diferentes, debe tener $m$ entradas de control. Cada combinación de las entradas de control indicará a la UAL que haga una operación concreta. Por ejemplo, la UAL de la figura 30 puede hacer cuatro operaciones.

**4.** Un cierto número de salidas de un bit, que llamamos ***bits de estado***, y tienen la función de indicar algunas circunstancias que se pueden haber producido durante el cálculo.

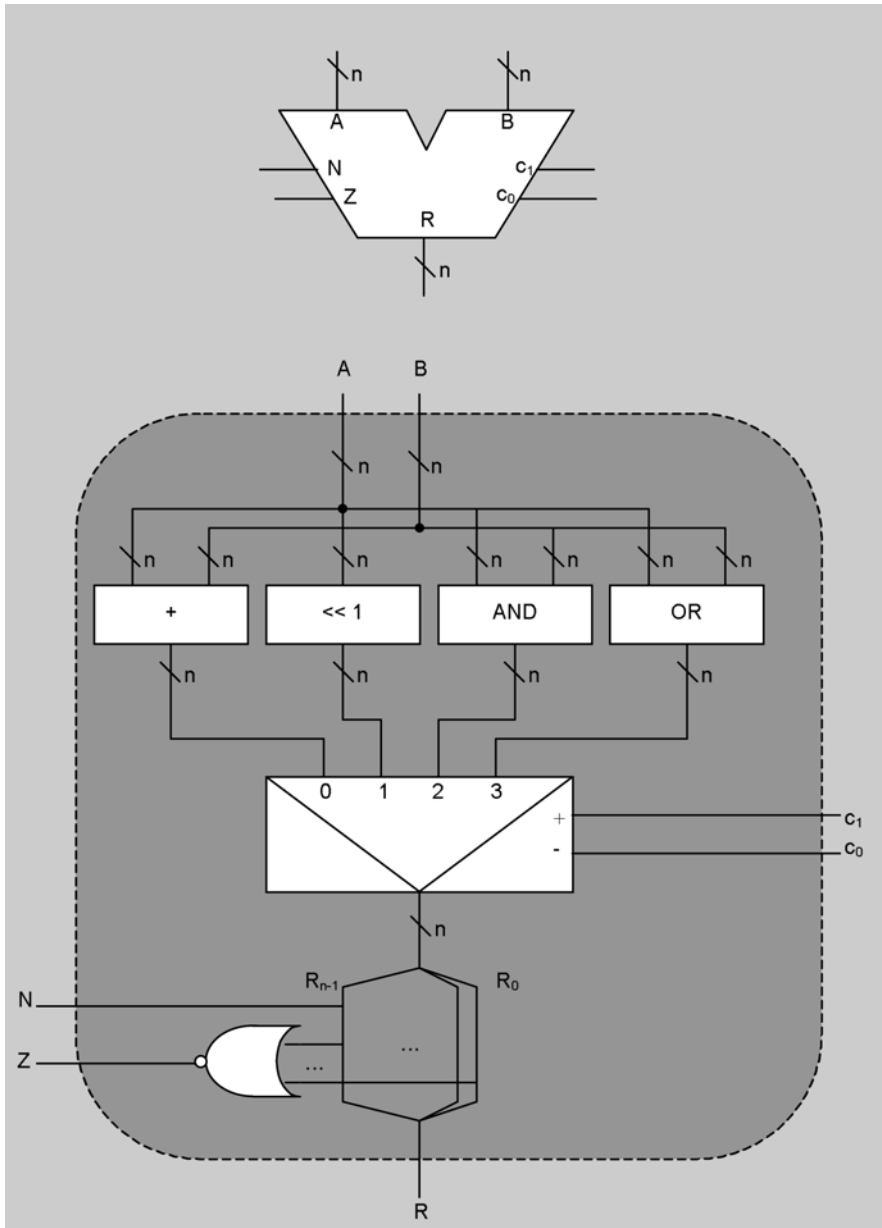
El bit de desbordamiento se suele identificar con las letras *O* o *V*, que derivan de la palabra inglesa ***overflow***, que significa 'desbordamiento'.

Los bits de estado más habituales son los que indican que se ha producido acarreo en el último bit ($C$), si se ha producido desbordamiento ($V$), si el resultado de la operación ha sido negativo ($N$) o si el resultado de la operación ha sido 0 ($Z$).

La figura 30 muestra la representación gráfica y la implementación interna de una UAL que genera los bits de estado $N$ y $Z$ y que realiza las siguientes operaciones:

| $c_1$ | $c_0$ | $R$ |
|---|---|---|
| 0 | 0 | $A + B$ |
| 0 | 1 | $2 * A$ |
| 1 | 0 | $a$ AND $b$ |
| 1 | 1 | $a$ OR $b$ |

Figura 30




**Nota**

En esta figura hemos disgregado los bits del bus correspondiente a la señal de salida R para dibujar la implementación de los bits *N* y *Z*. Se tiene que interpretar que todos los bits de *R* se conectan a la puerta NOR (de *n* entradas) que computa *Z*.

## Actividades

**60.** Contestad los siguientes apartados:

**a)** Diseñad una UAL que, a partir de dos entradas de control $c_1$ y $c_0$, realice las siguientes operaciones sobre dos números A y B de cuatro bits:

- $[c_1\ c_0] = [0\ 0] : R = A + B$.
- $[c_1\ c_0] = [0\ 1] : R = A - B$.
- $[c_1\ c_0] = [1\ 0] : R = A$.
- $[c_1\ c_0] = [1\ 1] : R = -A$.

**b)** Añadid a la UAL diseñada en el apartado anterior los circuitos necesarios para calcular los siguientes bits de condición:

- *Vb*: desbordamiento si se interpreta que los números de entrada están codificados en binario.
- *V*: desbordamiento si se interpreta que los números de entrada están codificados en complemento a 2.
- *N:* $N = 1$ si el resultado de la operación interpretado en complemento a 2 es negativo, y 0 en el caso contrario.
- *Z:* $Z = 1$ si el valor de la salida es 0, y $Z = 0$ en el caso contrario.

## Resumen

En este módulo se han estudiado los fundamentos de los circuitos electrónicos digitales. Se ha visto que se construyen a partir de los mismos elementos que la lógica natural, formalizada matemáticamente por el álgebra de Boole: los elementos básicos son los valores 0 y 1 y las operaciones suma lógica, producto lógico y negación. A partir de aquí se han introducido las variables y funciones lógicas.

Se han expuesto dos maneras de representar funciones lógicas: las expresiones algebraicas y las tablas de verdad. Se ha visto que son especialmente cómodas las expresiones en suma de productos.

Después se ha visto cómo se implementan físicamente las operaciones lógicas básicas para construir circuitos mediante las puertas lógicas. Se ha aprendido la manera de minimizar circuitos a dos niveles, mediante el método de minimización de Karnaugh.

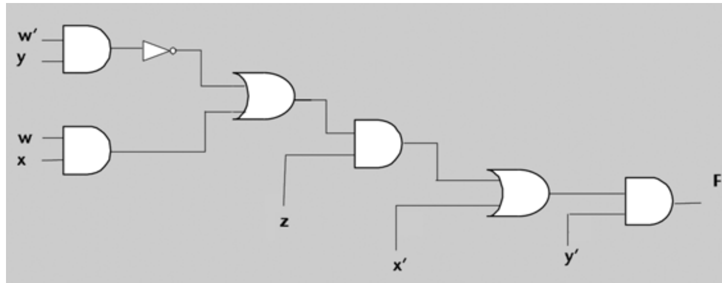
Finalmente, se han conocido varios bloques combinacionales, y se ha aprendido a utilizar la funcionalidad de cada uno de ellos para diseñar y entender circuitos complejos.

## Ejercicios de autoevaluación

Entre paréntesis se indica el apartado de los apuntes que se debe haber estudiado para resolver cada ejercicio.

**1. (1.2)** Demostrad que las leyes de De Morgan también se cumplen para tres y cuatro variables. Es suficiente con demostrar que $(xyz)' = x' + y' + z'$ y que $(xyzw)' = x' + y' + z' + w'$, porque las otras leyes se obtienen directamente aplicando el principio de dualidad.

**2. (2.1)** Analizad el siguiente circuito combinacional y expresad algebraicamente en forma de suma de productos la función que implementa:




**3. (2.2)** Diseñad a dos niveles un circuito combinacional que permita multiplicar dos números enteros de dos bits representados en complemento a 2.

**4. (2.3)** Sintetizad de manera mínima a dos niveles el circuito descrito en la actividad 24.

**5. (3.1)** Diseñad un multiplexor 16-1 utilizando únicamente multiplexores 4-1 (sin utilizar ninguna puerta lógica adicional).

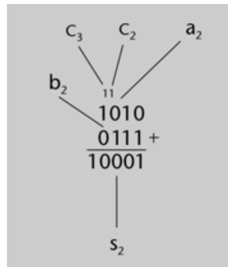
**6. (3.2)** Diseñad un descodificador 4-16 utilizando únicamente cinco descodificadores 2-4.

**7. (3.2)** Suponed que $X$ es un número natural codificado con tres bits. Implementad la función $(3 \cdot X)$ mod 8 usando sólo un descodificador 3-8 y un codificador 8-3.

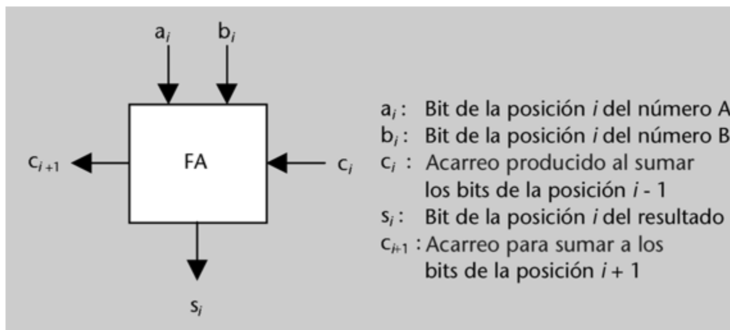
**8. (3.3)** Resolved los siguientes apartados:
**a)** Implementad un *desplazador programable a la izquierda* de números de ocho bits. El desplazador tiene una entrada de control de tres bits, $C$, que indica el número de bits del desplazamiento.
**b)** Si $C$ codifica en binario el valor $n$, ¿a qué operación aritmética equivale este desplazamiento?
**c)** Suponed que $C$ = 010 e indicad cuándo se produciría desbordamiento en los casos de interpretar la entrada como un número natural o bien como uno entero representado en complemento a 2.
Supongamos ahora que queremos hacer la operación $X / 2^n$ usando un desplazador en la derecha.
**d)** ¿Qué tipo de desplazador tenemos que utilizar si $X$ es un número natural representado en binario? ¿Y si es un número entero representado en complemento a 2?
**e)** ¿En qué casos el resultado de la división no es exacto?

**9. (3.6)** Sintetizad un comparador de números enteros de 4 bits codificados en complemento a 2, utilizando comparadores de números naturales de cuatro bits y las puertas lógicas necesarias.

**10. (3.7)** Cuando hacemos una suma de números binarios, la hacemos bit a bit (de la misma manera que en decimal la hacemos dígito a dígito). Para obtener el bit de la posición $i$ necesitamos conocer los bits que se encuentran en la posición $i$ en los dos operandos y el acarreo que se genera en la suma de los bits de la posición $i - 1$.
La suma de estos 3 bits da como resultado 2 bits: el bit $s_i$, que corresponde al bit de la posición $i$ de la suma, y el bit $c_i + 1$, que es el acarreo que se genera para la suma de los bits de la posición $i + 1$.
A continuación se muestra un ejemplo para $A$ = 1010 y $B$ = 0111. El ejemplo ofrece concretamente la suma de los bits de la posición 2 (recordad que el bit más a la derecha es el de la posición 0).




El sumador de números de un bit tiene tres entradas ($a_i$, $b_i$ y $c_i$) y dos salidas ($s_i$ y $c_{i+1}$). Este sumador recibe el nombre de *full adder* (sumador completo), y se abrevia FA. La figura siguiente muestra la estructura.




**a)** Escribid la tabla de verdad y realizad la implementación interna (mediante puertas lógicas) de un FA como el descrito.

**b)** Utilizad cuatro FA para construir un sumador de dos números de cuatro bits. ¿Qué hay que conectar a la entrada que corresponde al bit $c_0$?

**11. (3.7)** Diseñad un circuito que sea capaz de sumar o restar dos números codificados en complemento a 2 con 4 bits de acuerdo con lo que valga la señal de entrada *s'/r*: si vale 0, el circuito debe realizar la suma, y si vale 1, debe realizar la resta. El circuito debe generar además una señal de salida *C* que indique si se ha producido acarreo en el último bit.

**12.(3.7)** Queremos diseñar un circuito que controle los ascensores de un edificio de cinco plantas y planta baja. El edificio tiene dos ascensores, *A* y *B*, cada uno con dos señales asociadas:

- Una señal que indica en binario en qué planta está el ascensor en cada momento (*plantaA* y *plantaB*).

- Una señal de activación (*actA*, *actB*).

Cuando una señal de activación se pone a 1, el ascensor correspondiente se pone en movimiento; la señal *destino* le indica, en binario (número natural), hacia qué planta debe ir. Después de que el ascensor se ponga en movimiento, la señal de activación vuelve a 0 automáticamente (no es preciso que nos preocupemos de hacerlo).

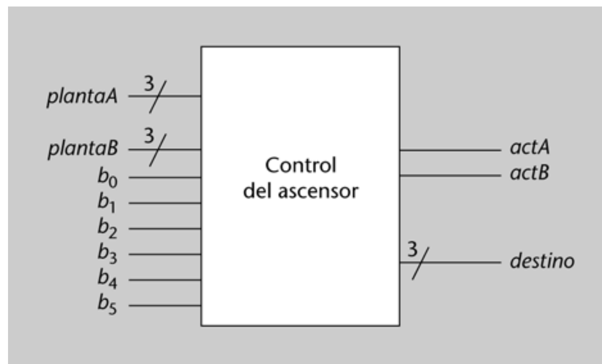
En cada planta *i* (*i* = 0, 1, ..., 5) hay un botón para llamar al ascensor, conectado a la señal $b_i$. Cuando alguien lo llama desde la planta *i*-ésima, la señal $b_i$ se pone a 1 y vuelve a 0 cuando el ascensor ha llegado a la planta correspondiente.

Mientras no haya ninguna llamada, los ascensores deben permanecer detenidos. Cuando alguien llame al ascensor desde una planta, se debe mover el que esté más cerca. Si los dos están a la misma distancia, irá el ascensor *A*.

Para facilitar el diseño, supondremos algunas simplificaciones en el sistema:

- Nunca se pulsará más de un botón de llamada al mismo tiempo.

- En el momento de pulsar un botón de llamada, los dos ascensores están detenidos.




Diseñad el circuito de control de los ascensores utilizando los bloques y las puertas que sean necesarios, y teniendo cuidado de dar a todos los buses el ancho mínimo necesario. Si queréis, podéis utilizar como bloques los circuitos que se diseñan en las actividades 56 y 57, dimensionando la amplitud de las entradas y salidas según os sea necesario.

**13. (3.8)** Diseñad una UAL con salida R de 4 bits que, a partir de tres entradas de control $c_2$, $c_1$ y $c_0$, realice las operaciones siguientes sobre dos números *A* y *B* de 4 bits:

- $[c_2\, c_1\, c_0] = [0\ x\ x] : R = B$.
- $[c_2\, c_1\, c_0] = [1\ 0\ 0] : R = A + B$.
- $[c_2\, c_1\, c_0] = [1\ 0\ 1] : R = A - B$.
- $[c_2\, c_1\, c_0] = [1\ 1\ 0] : R = B >> 1$.
- $[c_2\, c_1\, c_0] = [1\ 1\ 1] : R = A$ AND $B$.

# Solucionario

## Actividades

**1.** Para evaluar la expresión $x + y \cdot (z + x')$ sustituiremos las variables para cada combinación de valores:

Para el caso de $[x\ y\ z] = [0\ 1\ 0]$, tenemos lo siguiente:
$$0 + 1 \cdot (0 + 1) = 0 + 1 \cdot 1 = 0 + 1 = 1.$$
Para el caso $[x\ y\ z] = [1\ 1\ 0]$, tenemos lo siguiente.
$$1 + 1 \cdot (0 + 0) = 1 + 1 \cdot 0 = 1 + 0 = 1.$$
Para el caso $[x\ y\ z] = [0\ 1\ 1]$, tenemos lo siguiente:
$$0 + 1 \cdot (1 + 1) = 0 + 1 \cdot 1 = 0 + 1 = 1.$$

**2.** Para la demostración, sustituiremos las variables de entrada por todas las combinaciones de valores que puedan tomar. Las igualdades se deben cumplir para todos los casos.

- Igualdad $(x + y)' = x' \cdot y'$
- Caso $x = 0$ y $y = 0$:
  $(x + y)' = (0 + 0)' = 0' = 1$,
  $x' \cdot y' = 0' \cdot 0' = 1 \cdot 1 = 1$.
- Caso $x = 0$ y $y = 1$:
  $(x + y)' = (0 + 1)' = 1' = 0$,
  $x' \cdot y' = 0' \cdot 1' = 1 \cdot 0 = 0$.
- Caso $x = 1$ y $y = 0$:
  $(x + y)' = (1 + 0)' = 1' = 0$,
  $x' \cdot y' = 1' \cdot 0' = 0 \cdot 1 = 0$.
- Caso $x = 1$ y $y = 1$:
  $(x + y)' = (1 + 1)' = 1' = 0$,
  $x' \cdot y' = 1' \cdot 1' = 0 \cdot 0 = 0$.

Puesto que las dos expresiones valen lo mismo para todos los casos posibles, concluimos que son equivalentes.

- Igualdad $(x \cdot y)' = x' + y'$
- Caso $x = 0$ y $y = 0$:
  $(x \cdot y)' = (0 \cdot 0)' = 0' = 1$,
  $x' + y' = 0' + 0' = 1 + 1 = 1$.
- Caso $x = 0$ y $y = 1$:
  $(x \cdot y)' = (0 \cdot 1)' = 0' = 1$,
  $x' + y' = 0' + 1' = 1 + 0 = 1$.
- Caso $x = 1$ y $y = 0$:
  $(x \cdot y)' = (1 \cdot 0)' = 0' = 1$,
  $x' + y' = 1' + 0' = 0 + 1 = 1$.
- Caso $x = 1$ y $y = 1$:
  $(x \cdot y)' = (1 \cdot 1)' = 1' = 0$,
  $x' + y' = 1' + 1' = 0 + 0 = 0$.

Puesto que las dos expresiones valen lo mismo para todos los casos posibles, concluimos que son equivalentes.

**3.** La ley 4 de álgebra de Boole dice que $x + 1 = 1$ y que $x \cdot 0 = 0$.

Demostraremos sólo la primera igualdad, la segunda quedará demostrada por el principio de dualidad.

El axioma *e* dice que $x + x' = 1$. Por tanto,

$$x + 1 = x + x + x'.$$

La ley 2 (de idempotencia) dice que $x + x = x$. Por tanto,

$$x + x + x' = x + x' = 1.$$

Hemos obtenido la última igualdad aplicando de nuevo el axioma *e*. Por tanto, $x + 1 = 1$, tal como queríamos demostrar.

**4.**
- La función $g_1$ vale 1 sólo cuando $x$ vale 1 e $y$ vale 1. Por tanto, una posible expresión lógica para esta función es $g_1(x, y) = x \cdot y$.
- La función $g_3$ vale 1 cuando $x$ vale 1 e $y$ vale 0, o bien cuando $x$ vale 1 e $y$ vale 1. Una posible expresión por función es, por tanto, $x \cdot y' + x \cdot y$. También podemos apreciar que la

función vale 1 cuando la variable $x$ vale 1, independientemente del valor que tome la variable $y$. Por tanto, una posible expresión lógica para esta función es $g_3(x, y) = x$.
- La función $g_6$ vale 1 cuando $x$ vale 1 e $y$ vale 0, o bien cuando $x$ vale 0 e $y$ vale 1. Por tanto, otra posible expresión lógica para esta función es $g_6(x, y) = xy' + x'y$.
- La función $g_7$ vale 1 cuando $x$ vale 0 e $y$ vale 1, cuando $x$ vale 1 e $y$ vale 0, o bien cuando $x$ vale 1 e $y$ vale 1. Por tanto, una posible expresión lógica para esta función es $g_7(x, y) =$ $= x'y + xy' + xy$. También podemos decir que la función vale 1 siempre que no se cumpla que x = 0 e y = 0. Por tanto, otra expresión para la función es $g_7(x, y) = (x'y')'$. Si aplicamos la segunda ley de De Morgan sobre esta expresión, y después de la ley de involución, obtenemos la expresión $g_7(x, y) = x'' + y'' = x + y$.
- La función $g_{10}$ vale 1 cuando $x$ vale 0 e $y$ vale 0, o bien cuando $x$ vale 1 e $y$ vale 0. Es decir, la función vale 1 cuando la variable $y$ vale 0, independientemente del valor de $x$. Por tanto, la función vale lo contrario de lo que valga $y$, y una posible expresión lógica para esta función es $g_{10}(x, y) = y'$.

**5.** Tenemos la expresión:

$$wx + xy' + yz + xz' + xy.$$

Para simplificarla aplicaremos la propiedad distributiva. Agrupamos el segundo término con el último, y obtenemos lo siguiente:

$$wx + x(y' + y) + yz + xz'.$$

Por el axioma de complementación sabemos que $y + y' = 1$. Si hacemos la sustitución en la expresión y aplicamos el axioma de los elementos neutros, obtenemos lo siguiente:

$$w \cdot x + x \cdot 1 + yz + xz' = wx + x + yz + xz'.$$

Por la ley de absorción, $wx + x = x$. Por tanto,

$$wx + x + yz + xz' = x + yz + xz'.$$

Aplicamos de nuevo la ley de absorción a $x + xz'$, y obtenemos lo siguiente:

$$x + yz + xz' = x + yz.$$

Por tanto, concluimos con lo que reproducimos a continuación:

$$wx + xy' + yz + xz' + xy = x + yz.$$

**6.**
- Para la primera condición debemos encontrar una expresión que valga 1 cuando $x = 1$ o $y' = 1$. La expresión que vale 1 cuando se cumple esta condición es $x + y'$.
- La segunda condición es $x = 0$ y $z = 1$. Esto es lo mismo que decir que $x' = 1$ o $z = 1$. La expresión que vale 1 en este caso es $x' \cdot z$.
- La tercera condición, $x = 1$, $y = 1$ y $z = 1$, sólo se cumple por la expresión $x \cdot y \cdot z$.
- Dado que la función debe valer 1 en cualquiera de los tres casos, es decir, cuando se cumple la condición 1, la condición 2 o la 3, tenemos que la expresión de la función es la siguiente:

$$F = x + y' + x'z + xyz.$$

**7.** La función debe valer 1 cuando se tenga que activar la alarma. Según nos dice el enunciado, la alarma se activa cuando el interruptor general está cerrado ($ig = 0$) y alguna de las dos cajas fuertes está abierta (es decir, $x_A = 1$ o $x_B = 1$).

Una expresión para la función S que vale 1 cuando $ig = 0$ y ($x_A = 1$ o $x_B = 1$) es la siguiente:

$$s = ig'(x_A + x_B).$$

**8.** Para saber qué asignaturas ha aprobado Juan y cuál ha suspendido, plantearemos las afirmaciones de sus tres amigos de manera algebraica.

El resultado de cada asignatura se puede representar con una variable booleana. Esta variable valdrá 1 si la asignatura está aprobada y 0 en el caso contrario.

Las variables que utilizaremos para cada una de las asignaturas serán $m$ para matemáticas, $f$ para física y $q$ para química.

Entendemos que la "o" de estas frases es exclusiva. Es decir, la primera frase se podría sustituir por "o bien has aprobado matemáticas y has suspendido física, o bien has suspendido matemáticas y has aprobado física", y de manera similar con la segunda frase.

– Has aprobado matemáticas o física: $mf' + m'f$.
– Has suspendido química o matemáticas: $qm' + q'm$.
– Has aprobado dos asignaturas: $m'fq + mf'q + mfq'$.

Dado que se deben cumplir las tres afirmaciones a la vez, tenemos lo siguiente:

$$(mf' + m'f)\ (qm' + q'm)\ (m'fq + mf'q + mfq') = 1.$$

Simplificaremos la expresión aplicando los axiomas y teoremas del álgebra de Boole hasta que obtengamos la respuesta:

$$(mf' + m'f)(qm' + q'm)(m'fq + mf'q + mfq') =$$

(propiedad distributiva sobre los dos primeros paréntesis)

$$(mf'qm' + mf'q'm + m'fqm' + m'fq'm)(m'fq + mf'q + mfq') =$$

(complementación e idempotencia)

$$(0 + mf'q' + m'fq + 0)\ (m'fq + mf'q + mfq') =$$

(elementos neutros y distributiva sobre los dos paréntesis)

$$mf'q'm'fq + mf'q'mf'q + mf'q'mfq' + m'fqm'fq + m'fqmf'q + m'fqmfq' =$$

(complementación e idempotencia)

$$0 + 0 + 0 + m'fq + 0 + 0 = m'fq.$$

Hemos obtenido, por tanto, la siguiente expresión:

$$m'fq = 1.$$

Esta expresión sólo vale 1 cuando $m$ vale 0 y $f$ y $q$ valen 1, es decir, que Juan ha suspendido matemáticas y ha aprobado física y química.

**9.** Las tablas de verdad de estas funciones son las siguientes:

| $x$ | $y$ | $g_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $x$ | $y$ | $g_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $x$ | $y$ | $g_6$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $x$ | $y$ | $g_7$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $x$ | $y$ | $g_{10}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**10.** Pondremos las variables a la izquierda de la tabla, en orden decreciente de subíndices. Las cuatro variables pueden tomar dieciséis combinaciones de valores diferentes; las escribiremos en orden lexicográfico (si las interpretáramos como números naturales, corresponderían a los números desde el 0 hasta el 15). A la derecha de la tabla se encontrará la columna correspondiente a $f$.

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 0 | |
| 0 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 0 | 0 | |
| 1 | 1 | 0 | 1 | |
| 1 | 1 | 1 | 0 | |
| 1 | 1 | 1 | 1 | |

**11.** La tabla de verdad es la siguiente:

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**12.** Para obtener estas tablas, escribiremos una columna para cada uno de los términos producto, y después obtendremos la columna correspondiente a *f* haciendo la OR de las columnas parciales.

**a)**

| $x$ | $y$ | $z$ | $w$ | $x \cdot y' \cdot z \cdot w'$ | $y \cdot z' \cdot w$ | $x' \cdot z$ | $x \cdot y \cdot w$ | $x' \cdot y \cdot z' \cdot w'$ | $f$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**b)**

| $x$ | $y$ | $z$ | $w$ | $x \cdot y$ | $x \cdot y + z$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

**13.** Escribiremos las tablas de verdad de las dos funciones:

| $x$ | $y$ | $z$ | $w$ | $x' \cdot y'$ | $y \cdot w$ | $x' \cdot w$ | $f$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

| $x$ | $y$ | $z$ | $w$ | $x' \cdot y' \cdot z' \cdot w'$ | $x' \cdot z' \cdot w$ | $y \cdot z' \cdot w$ | $x \cdot y \cdot w$ | $x' \cdot z$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

Deducimos que las funciones no son equivalentes porque sus tablas de verdad son diferentes.

**14.** La tabla de verdad es la siguiente:

| $ig$ | $x_A$ | $x_B$ | $s$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

**15.** Para resolver este problema, asignaremos una función a cada una de las afirmaciones de los amigos de Juan. La función valdrá 1 para las combinaciones que hacen que la afirmación sea cierta, y 0 en el caso contrario. Las funciones que representan las afirmaciones de los amigos corresponden a las funciones $F_1$, $F_2$ y $F_3$, respectivamente.
Llamaremos $m$, $f$ y $q$ las variables que representan las asignaturas. Estas variables valdrán 1 cuando la asignatura esté aprobada, y 0 si está suspendida.

La solución representada por la función $F$ se obtiene haciendo un AND de las funciones $F_1$, $F_2$ y $F_3$, ya que las tres frases son ciertas simultáneamente. La combinación por la que $F$ vale 1 corresponderá a la solución del problema.

| $m$ | $f$ | $q$ | $F_1$ | $F_2$ | $F_3$ | $F$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

La función $F$ indica que Juan ha aprobado física y química y ha suspendido matemáticas.

**16.**
$f_0 = x_2'x_1x_0' + x_2x_1x_0$,
$f_1 = x_2x_1'x_0' + x_2x_1'x_0 + x_2x_1x_0' + x_2x_1x_0$,
$f_2 = x_2'x_1'x_0 + x_2'x_1x_0 + x_2x_1'x_0 + x_2x_1x_0$,
$f_3 = x_2'x_1'x_0 + x_2'x_1x_0' + x_2x_1'x_0' + x_2x_1x_0$.

**17.** Escribiremos la tabla de verdad y a partir de ésta obtendremos la expresión en suma de mintérminos.

**a)** La tabla de verdad de la función es la siguiente:

| $x$ | $y$ | $z$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

La expresión de $f$ en suma de mintérminos es ésta:

$$f = x'y'z + x'yz + xy'z + xyz' + xyz.$$

**b)** La tabla de verdad de la función es la siguiente:

| $x$ | $y$ | $z$ | $w$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

La expresión de $f$ en suma de mintérminos es la que reproducimos a continuación:

$$f = x'y'z'w' + x'y'z'w + x'y'zw + x'yz'w + xy'z'w + xy'zw + xyz'w.$$

**18.** Las combinaciones de entrada 101, 110 y 111 no se pueden producir y, por tanto, no nos importará el valor que tomen las salidas en estos casos.
La tabla de verdad de las funciones de este sistema es la siguiente:

| $x_2$ | $x_1$ | $x_0$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | x | x | x | x | x |
| 1 | 1 | 0 | x | x | x | x | x |
| 1 | 1 | 1 | x | x | x | x | x |

**19.**
**a)** A partir del enunciado, obtenemos que las expresiones de las funciones $R$ y $E$ son las siguientes:

$$R = h_0' + h_1't$$
$$E = t'h_0$$

Ahora bien, las combinaciones con $h_1 = 1$ y $h_0 = 0$ no se pueden producir (debemos suponer que los sensores funcionan bien). Por tanto, la tabla de verdad del sistema es la siguiente:

| $t$ | $h_1$ | $h_0$ | $R$ | $E$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | x | x |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | x | x |
| 1 | 1 | 1 | 0 | 0 |

**20.** Para realizar el análisis, es necesario obtener la expresión de la función a partir del circuito. A partir de la expresión obtendremos la tabla de verdad. Para rellenarla cómodamente, encontraremos previamente una expresión simplificada de la función.

**a)** La expresión que obtenemos directamente a partir del circuito es la siguiente:

$$F = (y' + w')\ (y + w) + (x' + z)\ (x + z').$$

Si simplificamos esta expresión, obtenemos lo siguiente:

$$F = y'w + yw' + x'z' + xz.$$

La tabla de verdad es la siguiente:

| $x$ | $y$ | $z$ | $w$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |

| x | y | z | w | f |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**b)** La expresión que obtenemos directamente a partir del circuito es la siguiente:

$$F = ((x' + w)\ (y + w'))'z + (y + w')z'.$$

Si simplificamos esta expresión, obtenemos lo que reproducimos a continuación:

$$\begin{aligned} F &= ((x' + w)' + (y + w')')z + (y + w')z' = \\ &= (xw' + y'w)z + (y + w')z' = \\ &= xzw' + y'zw + yz' + z'w'. \end{aligned}$$

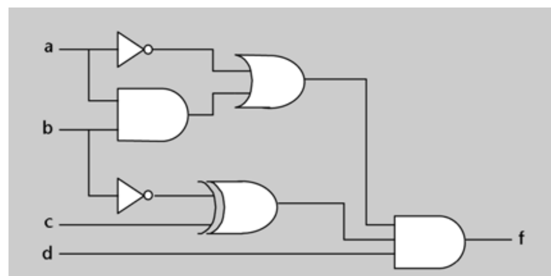
La tabla de verdad es la siguiente:

| x | y | z | w | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**21.** Para hacer la síntesis, simplemente dibujamos una línea para cada variable de la función y sustituimos las operaciones lógicas de la expresión por la puerta lógica correspondiente. La función es ésta:

$$f(a, b, c, d) = d{\cdot}(a' + a \cdot b){\cdot}(b' \oplus c).$$

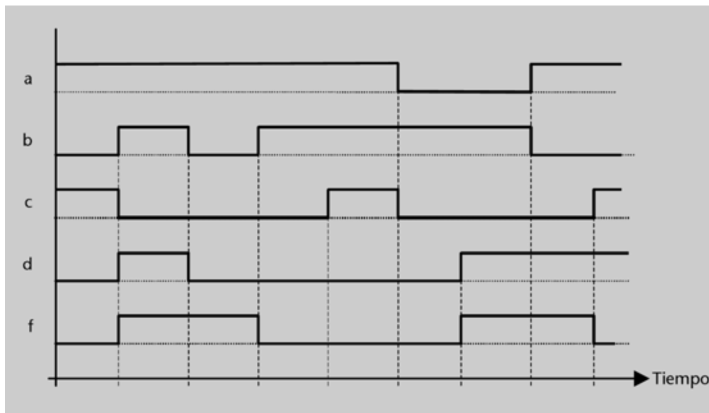
La siguiente figura muestra el circuito.




**22.** En la figura se muestra el cronograma. Dado que se trata de un circuito combinacional en el que no se consideran los retardos (tal como haremos habitualmente en esta asignatura), las variaciones en la salida se producen en el mismo momento en que se produce alguna

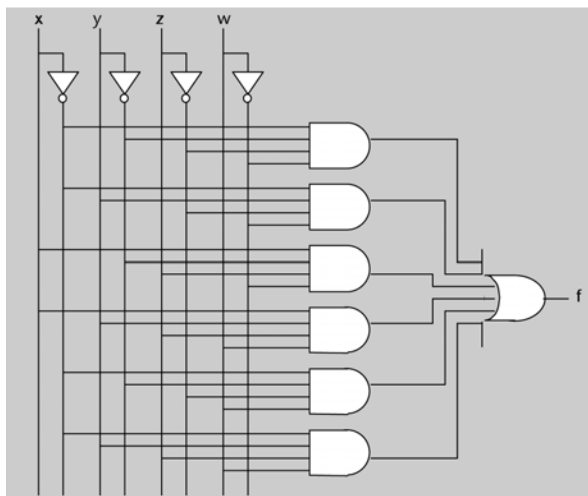
variación de las entradas (líneas punteadas verticales). Sin embargo, no todas las variaciones de las entradas producen una variación de la salida, como se puede ver en el cronograma.

La función que implementa el circuito es $f = b'c' + c'd$.

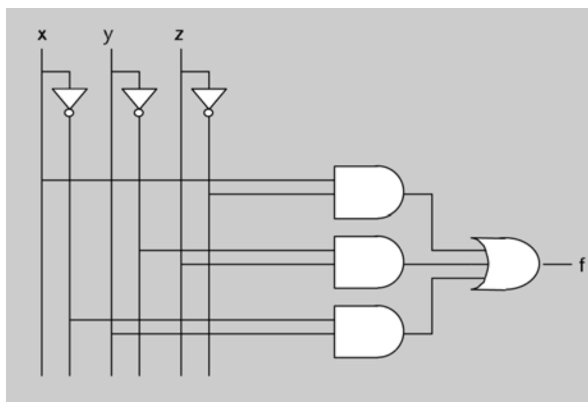



**23.** En las figuras se muestra la síntesis a dos niveles de cada circuito. Cada término *producto* está implementado por una puerta AND de tantas entradas como variables tiene el término *producto*. La suma lógica está implementada por una puerta OR de tantas entradas como términos *producto* tiene la expresión.

**a)**




**b)**




**24.**

**a)** El rango de valores que puede tomar un número natural de 2 bits es 0..3. Por tanto, el rango de la multiplicación de dos números será 0..9. Para representar el 9 necesitamos 4 bits. Por tanto, la salida tendrá 4 bits.

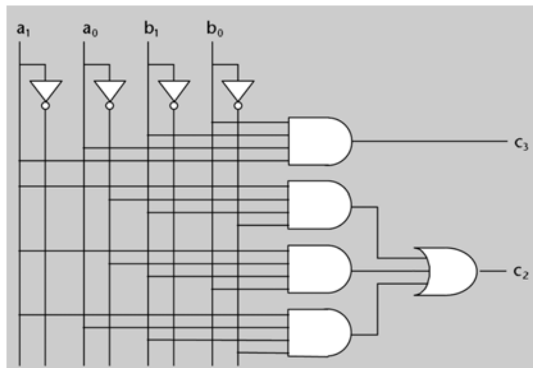
**b)** Denominaremos $A$ y $B$, respectivamente, a los números de entrada, y $C$ a su multiplicación. Para obtener la tabla de verdad pasamos $A$ y $B$ a decimal, calculamos $C = A \cdot B$ en decimal y, finalmente, codificamos $C$ en binario, con los bits $c_3 \dots c_0$. Por ejemplo, tomamos la fila $[a_1\ a_0\ b_1\ b_0] = [1\ 0\ 1\ 1]$. Tenemos que $A = 2$, $B = 3$. Por tanto, $C = 2 \cdot 3 = 6$, que se codifica $[0\ 1\ 1\ 0]$ en binario. La tabla de verdad completa es la siguiente:

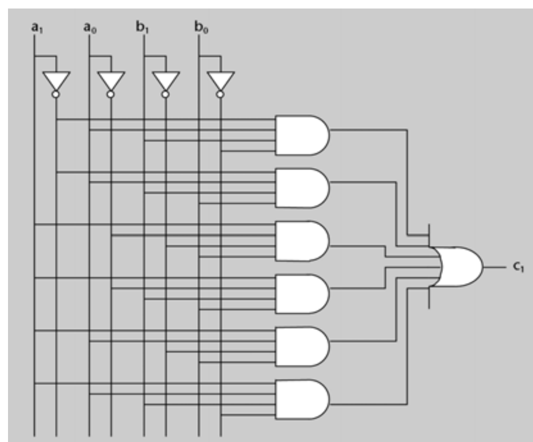
| $a_1$ | $a_0$ | $b_1$ | $b_0$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

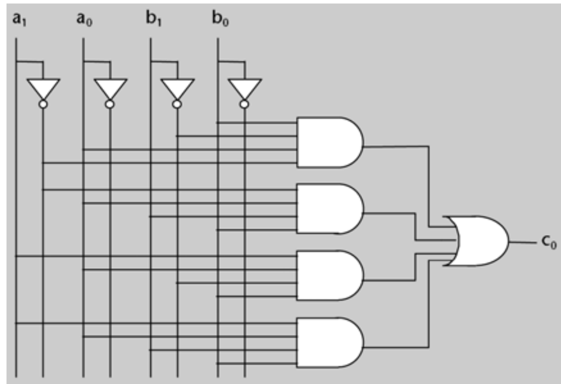
**c)** Las expresiones en suma de productos de las cuatro funciones de salida son las siguientes:

$c_3 = a_1a_0b_1b_0$,
$c_2 = a_1a_0'b_1b_0' + a_1a_0'b_1b_0 + a_1a_0b_1b_0'$,
$c_1 = a_1'a_0b_1b_0' + a_1'a_0b_1b_0 + a_1a_0'b_1'b_0 + a_1a_0'b_1b_0 + a_1a_0b_1'b_0 + a_1a_0b_1b_0'$,
$c_0 = a_1'a_0b_1'b_0 + a_1'a_0b_1b_0 + a_1a_0b_1'b_0 + a_1a_0b_1b_0$.

El circuito correspondiente a la función $C_3$ se puede implementar con una puerta AND. A continuación, se muestran los circuitos a dos niveles correspondientes a las tres funciones de salida restantes.



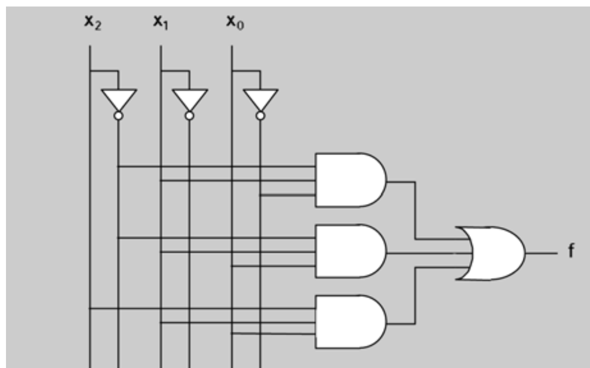






**25.** La expresión en suma de mintérminos de la función es la siguiente:

$$f = x_2'x_1x_0' + x_2'x_1x_0 + x_2x_1x_0$$

A continuación se muestra la síntesis a dos niveles de la función.




**26.** A continuación se muestra la tabla de verdad de la función y su minimización utilizando el método de Karnaugh.

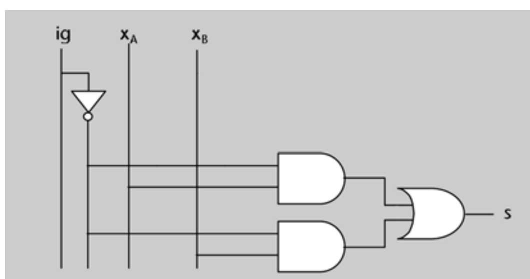
| ig | $x_A$ | $x_B$ | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

| $x_B$ \ ig $x_A$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |

Del rectángulo vertical se obtiene el término producto $ig' \cdot x_A$, y del rectángulo horizontal el término $ig' \cdot x_B$. Por tanto, la expresión de la función es ésta:

$$s = ig'x_A + ig'x_B.$$

A continuación se muestra la síntesis mínima a dos niveles de la función.



Fijaos en que si sacamos ig′ factor común en la expresión mínima de s, obtenemos $s = ig'(x_A + x_B)$, que es la expresión que hemos deducido en la actividad 7 (pero no es una suma de productos).

**27.** A continuación se muestran la tabla de verdad de la función y su minimización por Karnaugh.

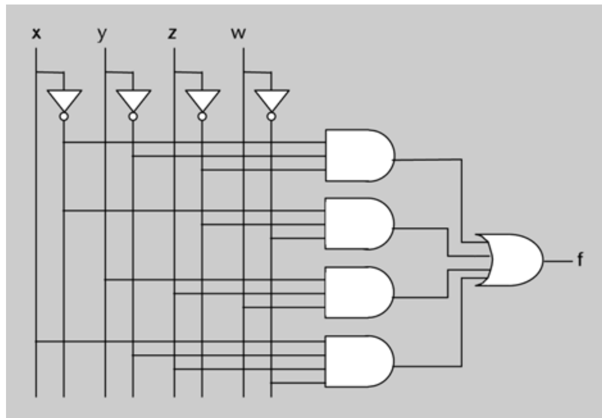
| x | y | z | w | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| zw \ xy | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 0 | 0 | 0 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 1 |

Del mapa de Karnaugh se deduce esta expresión:

$$f = x'y'z' + x'z'w' + yzw + xy'zw'.$$

El circuito mínimo a dos niveles se presenta en la siguiente figura. Se puede comprobar que es más sencillo que el que se ha obtenido en la actividad 23.




**28.** En la siguiente figura se muestra la tabla de verdad de las funciones y su minimización por Karnaugh.

| t | $h_1$ | $h_0$ | R | E |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | x | x |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | x | x |
| 1 | 1 | 1 | 0 | 0 |

R

| $h_0$ \ $th_1$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | x | x | 1 |
| 1 | 0 | 0 | 0 | 1 |

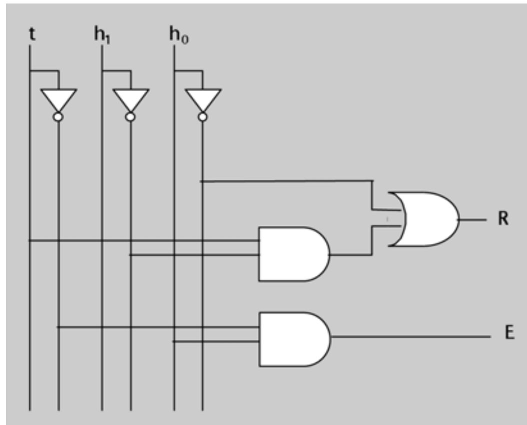
E

| $h_0$ \ $th_1$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | x | x | 0 |
| 1 | 1 | 1 | 0 | 0 |

Las funciones obtenidas son las siguientes:

$$R = h_0' + th_1',$$
$$E = t'h_0.$$

En la siguiente figura se muestra el circuito minimizado.

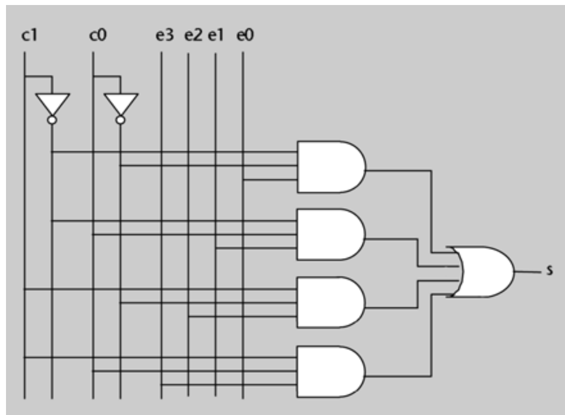



**29.** El valor de la salida *s* será el valor de $e_3$, $e_2$, $e_1$ o $e_0$ de acuerdo con lo que valgan las entradas de control $c_1$ y $c_0$.

Por ejemplo, si [*c*1 *c*0] [1 0], la salida *s* valdrá 1 si $e_2 = 1$, y valdrá 0 si $e_2 = 0$ (es decir, valdrá $e_2$). En este caso podríamos escribir que la expresión de la salida es c1c0'e2. En cada momento, las entradas c1 y c2 valen alguna de estas cuatro combinaciones: [0,0], [0,1], [1,0] o bien [1 1]. Por tanto, sólo uno de los productos *c*1'*c*0', *c*1*c*0', *c*1'*c*0 y *c*1*c*0 puede valer 1 en cada momento. Podemos concluir que una expresión válida para *s* es la siguiente:

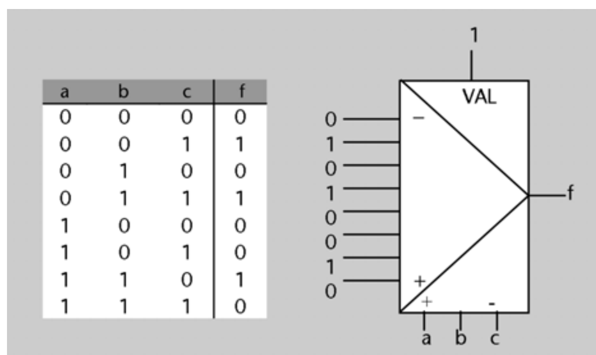
$$s = c1'c0'e0 + c1'c0e1 + c1c0'e2 + c1c0e3.$$

La implementación con puertas de esta expresión es la siguiente:



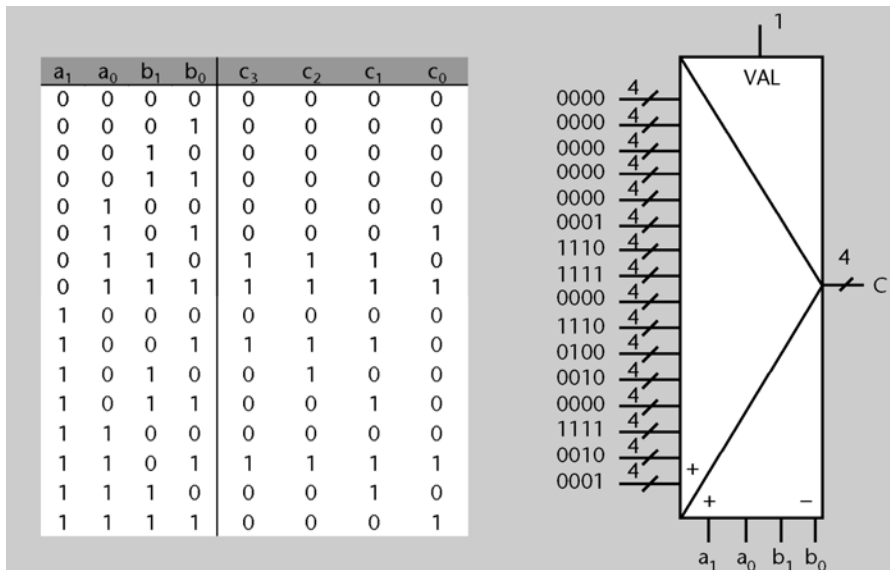

**30.** Para implementar esta función construiremos su tabla de verdad. A partir de ésta sabremos qué debemos conectar a las entradas del multiplexor para que en la salida nos aparezca un 0 o un 1 según indique la función.

En definitiva, se trata de traspasar la columna *f* de la tabla de verdad a las entradas del multiplexor.



| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

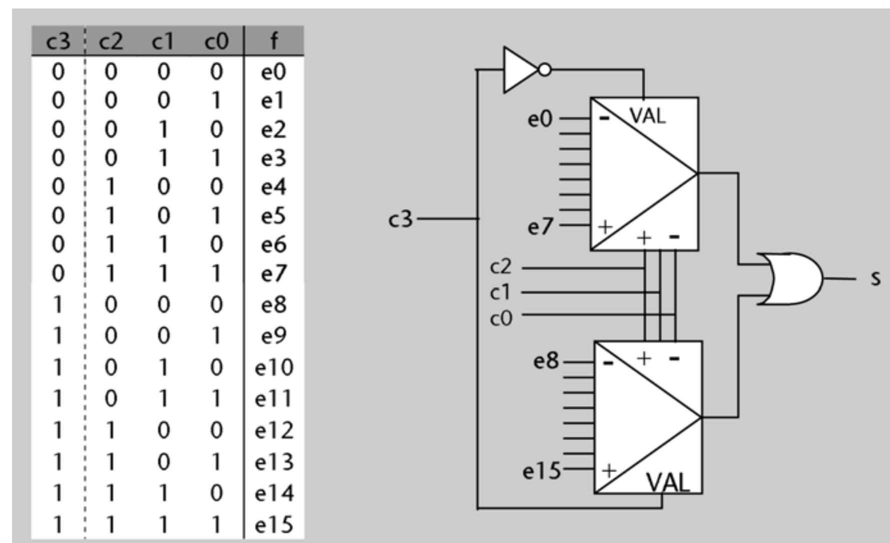
**31.** En primer lugar, construiremos la tabla de verdad del circuito.
El rango de un número entero representado en complemento a 2 con dos bits es [–2..1]. Por tanto, el rango de producto de dos de estos números es [–2..4.]. Para representar números dentro de este rango es suficiente con cuatro bits, ya que con éstos podemos representar números en el rango [–8..7].
Por tanto, la tabla de verdad tiene cuatro entradas y cuatro salidas. Las entradas son los dos bits de cada uno de los números *A* y *B*, y las salidas son los cuatro bits del resultado *C*.
Para implementar el circuito utilizaremos un multiplexor de buses de cuatro bits de dieciséis entradas de datos. La figura muestra esta implementación.



| $a_1$ | $a_0$ | $b_1$ | $b_0$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |



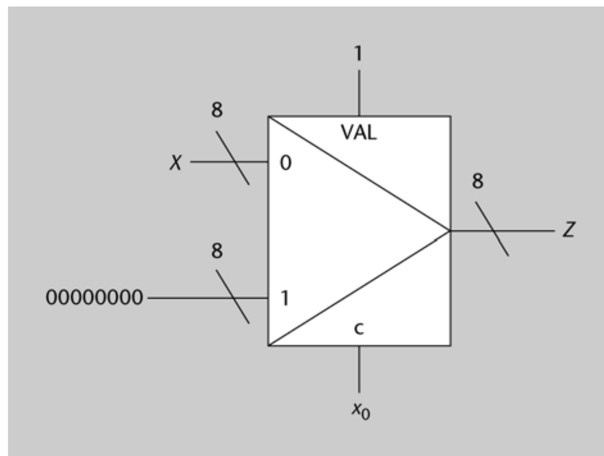
**32.** Escribimos primero la tabla de verdad correspondiente a un multiplexor 16-1. Éste tiene cuatro entradas de control (*c*3..*c*0) y dieciséis entradas de datos (*e*15..*e*0).
La línea discontinua en la tabla muestra que se puede descomponer el funcionamiento del circuito como si se tratara de dos multiplexores 8-1, los dos controlados por las señales *c*2, *c*1 y *c*0. La señal *c*3 determina cuál de los dos multiplexores funciona en cada momento.



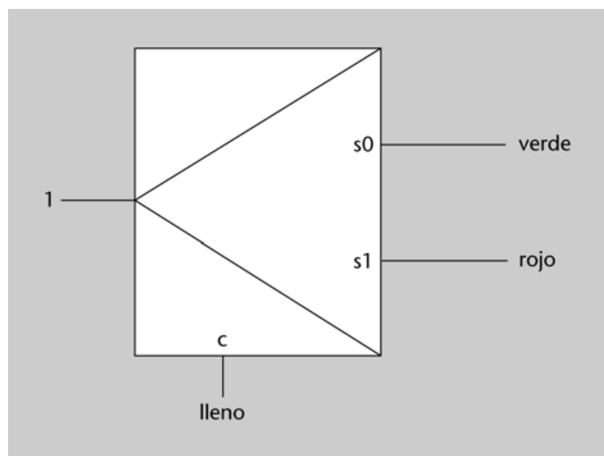
| c3 | c2 | c1 | c0 | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | e0 |
| 0 | 0 | 0 | 1 | e1 |
| 0 | 0 | 1 | 0 | e2 |
| 0 | 0 | 1 | 1 | e3 |
| 0 | 1 | 0 | 0 | e4 |
| 0 | 1 | 0 | 1 | e5 |
| 0 | 1 | 1 | 0 | e6 |
| 0 | 1 | 1 | 1 | e7 |
| 1 | 0 | 0 | 0 | e8 |
| 1 | 0 | 0 | 1 | e9 |
| 1 | 0 | 1 | 0 | e10 |
| 1 | 0 | 1 | 1 | e11 |
| 1 | 1 | 0 | 0 | e12 |
| 1 | 1 | 0 | 1 | e13 |
| 1 | 1 | 1 | 0 | e14 |
| 1 | 1 | 1 | 1 | e15 |



**33.** Puesto que *Z* debe valer uno de dos posibles valores, la podemos implementar mediante un multiplexor 2-1. El bit de menos peso de *X* nos indica si es par (0) o impar (1). Podemos, por lo tanto, conectar este bit a la entrada de control del multiplexor, y esto nos lleva a conectar *X* a la entrada de datos 0 y 00000000 a la entrada de datos 1.




**34.** Basta con poner un 1 a la entrada del demultiplexor, y hacer que la señal *llena* controle hacia cuál de las dos luces llega este 1.




**35.** Deduciremos las expresiones lógicas de cada una de las salidas y construiremos el circuito a partir de éstas. Para resolver la actividad, es conveniente tener delante la representación gráfica del codificador y su tabla de verdad (figura 21 de los apuntes).

Puesto que la salida del codificador debe codificar en binario la entrada de más peso que valga 1, podemos hacer el siguiente razonamiento:

**1)** La salida $s1$ se pondrá a 1 cuando estén en 1 las entradas $e3$ o $e2$ (en estos casos se debe codificar un 11 o un 10, respectivamente).

**2)** La salida $s0$ se pondrá a 1 cuando esté a 1 la entrada $e3$ (en este caso se tiene que codificar un 11, independientemente de los valores de las otras entradas) o bien cuando lo esté la entrada $e1$ y no lo esté ni la $e2$ ni la $e3$ (en este caso, [$e3$ $e2$ $e1$] = [0 0 1], se tiene que codificar un 01).

**3)** Por otro lado, la entrada de validación debe estar activa para que el codificador funcione como tal.

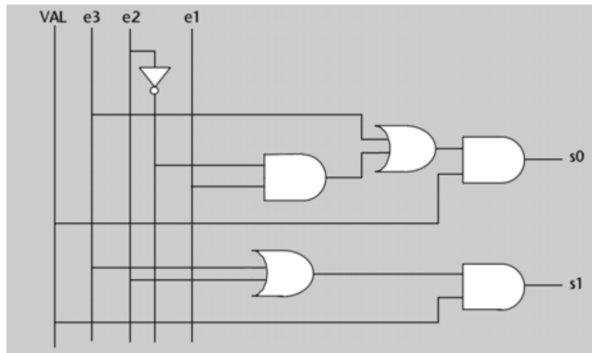
Por tanto, las expresiones algebraicas de las salidas son las siguientes:

– $s1$ valdrá 1 cuando VAL = 1 y $e3$ o $e2$ sean 1 (apartados 1 y 3):
$$s1 = \text{VAL}\ (e3 + e2).$$

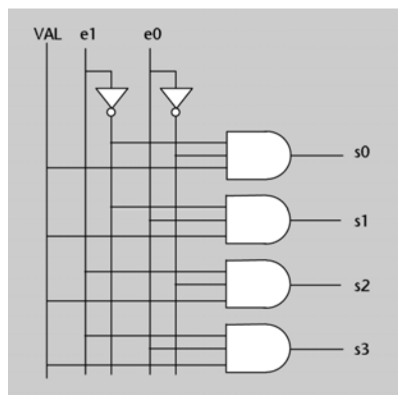
– $s0$ valdrá 1 cuando VAL = 1 y $e3$ sea 1, o cuando [$e3$ $e2$ $e1$] = [0 0 1] (apartados 2 y 3):
$$s0 = \text{VAL}\ (e3 + e3'e2'e1) = \text{VAL}\ (e3 + e2'e1).$$

A continuación se muestra una posible implementación de este codificador. Como se puede ver en la figura, el valor de la entrada $e0$ no tiene influencia en las salidas. Éste es el motivo

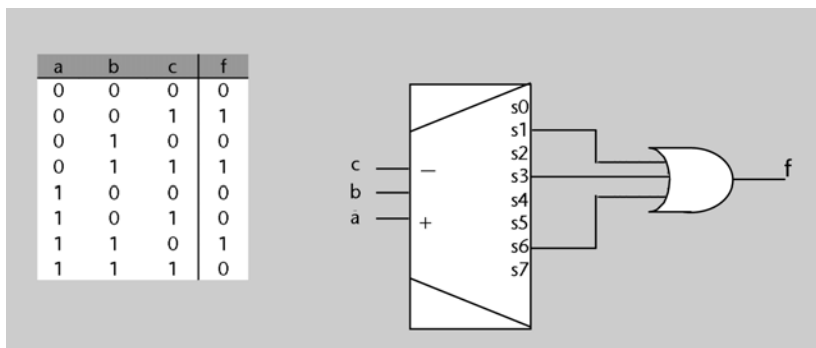
que hace que un codificador tenga un 0 en su salida tanto cuando tiene que codificar un 0 como cuando ninguna de sus entradas tiene un 1.




**36.** Cuando la entrada de validación está activa, cada salida de un descodificador se pone en 1 sólo cuando se produce una combinación determinada de las entradas. Por tanto, la implementación de cada una de las salidas será un circuito que detecte si esta combinación está presente en la entrada, tal como se muestra en la figura.




**37.** Primero haremos la tabla de verdad y, a partir de ésta, utilizaremos un descodificador para sintetizar el circuito. La puerta OR hace la suma lógica de los casos en que la función vale 1.



| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |



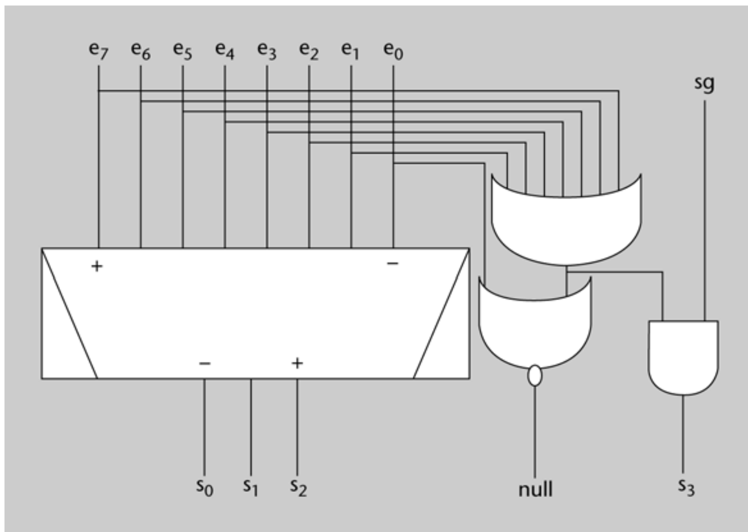
**38.** Podemos generar la magnitud del número usando un codificador 8-3, a cuyas entradas conectaremos *e*7, *e*6, ... *e*0. Las salidas de este codificador, por lo tanto, serán directamente las salidas *s*2, *s*1 y *s*0. Observemos que esto también hará que [*s*2 *s*1 *s*0] = [0 0 0] cuando no haya ninguna entrada *ei* a 1 (tal y como nos pide el enunciado).

A primera vista, podríamos pensar que el bit *s*3 (signo del número que hay que representar) lo podemos obtener directamente de la entrada *sg*, pero hay que tener en cuenta estas dos excepciones:

- cuando e0 = 1, entonces *s*3 debe valer 0 independientemente del valor de *sg*.
- cuando no haya ninguna entrada *ei* a 1, *s*3 también debe valer 0 independientemente del valor de *sg*.

Por lo tanto, *s*3 tiene que valer 1 cuando alguna de las entradas *e*7...e1 sea 1 y *sg* = 1.

La salida *null* debe valer 1 sólo cuando no haya ninguna entrada *ei* a 1.




**39.** Para deducir qué hace el circuito, construiremos su tabla de verdad. Ésta muestra también el valor de la entrada y la salida cuando las interpretamos como números representados en complemento a 2 (columnas *X* e *Y*).

| $X$ | $x_2$ | $x_1$ | $x_0$ | $y_2$ | $y_1$ | $y_0$ | $Y$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | –1 |
| 2 | 0 | 1 | 0 | 1 | 1 | 0 | –2 |
| 3 | 0 | 1 | 1 | 1 | 0 | 1 | –3 |
| –4 | 1 | 0 | 0 | 1 | 0 | 0 | –4 |
| –3 | 1 | 0 | 1 | 0 | 1 | 1 | 3 |
| –2 | 1 | 1 | 0 | 0 | 1 | 0 | 2 |
| –1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

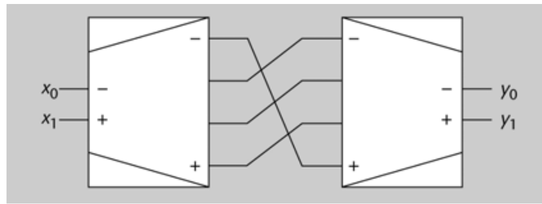
Como se puede apreciar en la tabla, lo que hace el circuito es cambiar el signo del número de la entrada. En el caso particular $X = –4$ esto no es posible, porque el 4 no se puede representar en complemento a 2 con sólo tres bits (el resultado es erróneo en este caso: se produce desbordamiento).

**40.** A partir de la codificación en binario de un número, debemos generar la codificación en binario de otro número. Lo podemos conseguir conectando las salidas de un descodificador con las entradas de un codificador, de manera que la salida *i* del descodificador se conecte con la entrada *j* del codificador, siendo *j* el valor que debe tener la salida cuando la entrada vale *i* (es decir, aplicando la misma técnica que en la actividad anterior).

La correspondencia entre valores de la entrada *X* y valores de la salida *Y* viene expresada por esta tabla:

| $X$ | $Y$ |
|---|---|
| 0 | 3 |
| 1 | 0 |
| 2 | 1 |
| 3 | 2 |

Por lo tanto, las conexiones se deben hacer tal y como se muestra en la siguiente figura.




**41.**

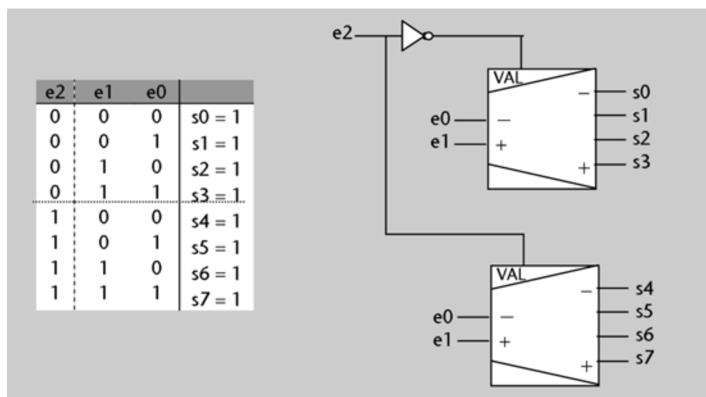
**a)** $E_0$ es 1 cuando las dos entradas del descodificador son 0, es decir, cuando $x_1 = 0$ y $x_0x_3 = 0$. Por tanto, la expresión algebraica para $E_0$ es $E_0 = x_1'(x_0x_3)'$. Si seguimos el mismo razonamiento para el resto de las salidas, obtenemos lo siguiente:

$E_0 = x_1'(x_0x_3)'$,
$E_1 = x_1'(x_0x_3) = x_1'x_0x_3$,
$E_2 = x_1(x_0x_3)'$,
$E_3 = x_1(x_0x_3) = x_1x_0x_3$.

**b)** La tabla de verdad se encuentra en la siguiente figura. Para obtener la columna correspondiente a *F* hemos dado un paso intermedio: hemos puesto en una columna el valor de *F* en términos de las entradas de datos del multiplexor ($E_i$), y en la columna final hemos sustituido las variables $E_i$ por el valor que toman para cada combinación de $x_i$, de acuerdo con las expresiones obtenidas en el apartado **a**.

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $F$ | $F$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $E_0$ | 1 |
| 0 | 0 | 0 | 1 | $E_0$ | 1 |
| 0 | 0 | 1 | 0 | $E_0$ | 0 |
| 0 | 0 | 1 | 1 | $E_0$ | 0 |
| 0 | 1 | 0 | 0 | $E_1$ | 0 |
| 0 | 1 | 0 | 1 | $E_1$ | 0 |
| 0 | 1 | 1 | 0 | $E_1$ | 0 |
| 0 | 1 | 1 | 1 | $E_1$ | 0 |
| 1 | 0 | 0 | 0 | $E_2$ | 0 |
| 1 | 0 | 0 | 1 | $E_2$ | 0 |
| 1 | 0 | 1 | 0 | $E_2$ | 1 |
| 1 | 0 | 1 | 1 | $E_2$ | 0 |
| 1 | 1 | 0 | 0 | $E_3$ | 0 |
| 1 | 1 | 0 | 1 | $E_3$ | 0 |
| 1 | 1 | 1 | 0 | $E_3$ | 0 |
| 1 | 1 | 1 | 1 | $E_3$ | 1 |

**42.** Esta actividad es muy similar a la actividad 32. En la siguiente figura se muestra la tabla de verdad del descodificador 3-8, que tiene tres entradas (*e*2..*e*0) y ocho salidas (*s*0..*s*7). Para simplificar la tabla, se ha puesto una única columna de salidas en la que hemos indicado cuál de éstas vale 1; todas las demás valen 0. Como se puede observar en la tabla de verdad, la entrada *e*2 determina cuál de los dos descodificadores 2-4 funciona en cada momento (y, por tanto, controla su entrada de validación). En cada descodificador 2-4, las entradas *e*1 y *e*0 determinan qué salida tiene que valer 1.



| e2 | e1 | e0 | |
|---|---|---|---|
| 0 | 0 | 0 | s0 = 1 |
| 0 | 0 | 1 | s1 = 1 |
| 0 | 1 | 0 | s2 = 1 |
| 0 | 1 | 1 | s3 = 1 |
| 1 | 0 | 0 | s4 = 1 |
| 1 | 0 | 1 | s5 = 1 |
| 1 | 1 | 0 | s6 = 1 |
| 1 | 1 | 1 | s7 = 1 |

**43.** Para resolver este ejercicio construiremos su tabla de verdad y después interpretaremos los números como naturales.
Llamamos $X$ al número que sale del codificador: $X = [x_1\ x_0]$..

| $A$ | $a_1$ | $a_0$ | $z$ | $x_1$ | $x_0$ | $X$ | $s_1$ | $s_0$ | $S$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 2 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 | 1 | 1 | 3 | 1 | 0 | 2 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 3 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 2 | 1 | 0 | 2 |
| 2 | 1 | 0 | 1 | 1 | 1 | 3 | 1 | 1 | 3 |
| 3 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Tal como podemos ver en la tabla,

$$X = (A + 1) \bmod 4.$$

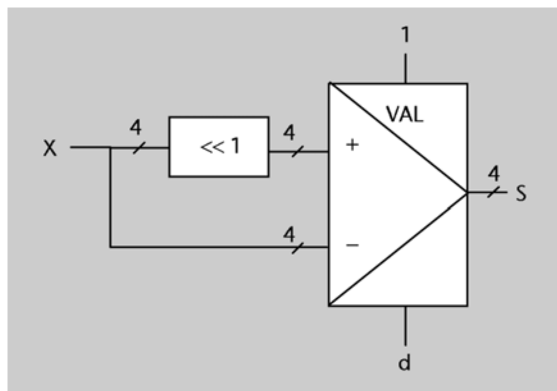
La variable $z$ selecciona entre las entradas $A$ y $X$, y, por tanto, la salida se puede expresar con la siguiente tabla:

| z | S |
|---|---|
| 0 | A |
| 1 | (A+1) mod 4 |

Esto se puede resumir en la siguiente expresión:

$$S = (A + z) \bmod 4.$$

**44.**
**a)** En la figura se muestra la implementación del circuito. La señal $d$ determina si la salida $S$ vale lo mismo que la entrada $X$ ($d = 0$) o $X$ desplazada un bit a la izquierda ($d = 1$).




**b)** Este desplazamiento de un bit a la izquierda es equivalente a multiplicar la entrada por 2. Por tanto, $S = 2 \cdot X$.

**c)** Se produce desbordamiento cuando el resultado no se puede representar con 4 bits.

– Si interpretamos los números $X$ y $S$ como números naturales codificados en binario, el rango de $X$ y $S$ es [0..15]. Por tanto, cuando $X$ sea mayor que 7, $S$ no podrá representar el resultado y se producirá desbordamiento.

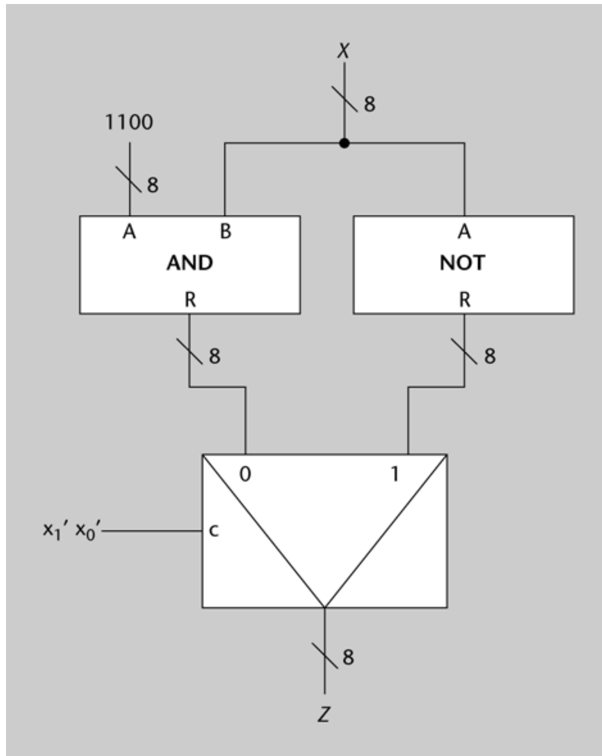
$$\text{Desbordamiento si } X > 7.$$

– Si interpretamos los números como enteros codificados en complemento a dos, el rango de $X$ y $S$ es [–8..7]. Por tanto, se producirá desbordamiento cuando $X$ sea menor que –4 o mayor que 3.

$$\text{Desbordamiento si } X < -4 \text{ o } X > 3.$$

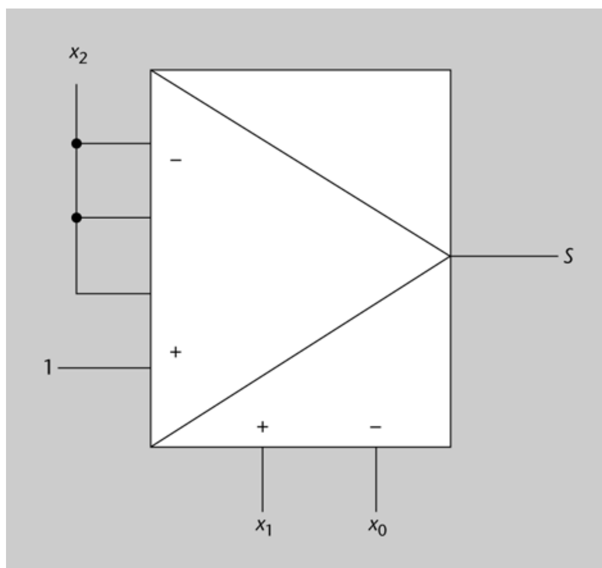
**45.** La salida $Z$ debe tomar uno de dos valores; esto lo podemos conseguir mediante un multiplexor 2-1. Cuál de los dos valores tome depende de si $X$ es múltiplo de 4 o no. Un número representado en binario es múltiplo de 4 si los dos bits de menor peso valen 0; por lo tanto, a la entrada de control del multiplexor debemos conectar $x_1'x_0'$.

Para obtener los valores que debemos conectar a cada una de las entradas de datos del multiplexor, podemos usar los bloques que se describen en el apartado 3.4, tal y como se muestra en la figura.




**46.**

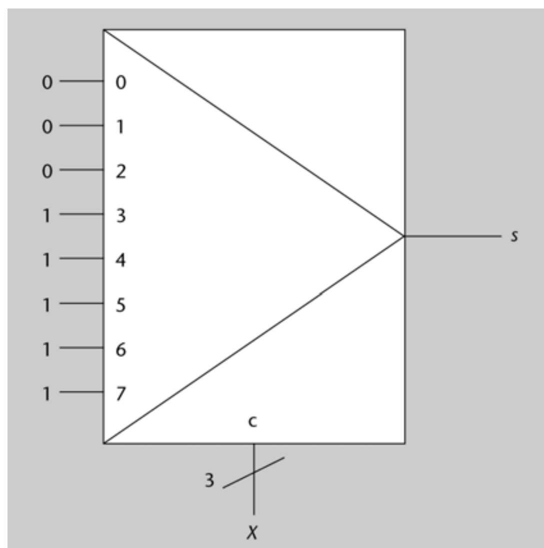
**a)** Vemos que para cualquier combinación de valores de $x_1$ y $x_0$ la señal de salida $s$ vale $x_2$, excepto en el caso $x_1 = x_0 = 1$ en el que $s$ vale 1 ($x_2 + x_2'$). Por lo tanto, podemos implementar $M0$ por ejemplo con un multiplexor 4-1 y sin ninguna puerta adicional. De esta manera:



Otra opción es implementar la función a partir de su tabla de verdad, tal y como se hace en la figura 18 del apartado 3.1:

| $x_2x_1x_0$ | $s$ |
|---|---|
| 000 | 0 |
| 001 | 0 |
| 010 | 0 |
| 011 | 1 |
| 100 | 1 |
| 101 | 1 |
| 110 | 1 |
| 111 | 1 |

En este caso utilizaremos un multiplexor 8-1, a las que conectaremos las variables de la función, y copiaremos los 1 y 0 en las entradas de datos del multiplexor.




**b)** El valor *U* es una XOR de *X* con la negación de sí misma. Dado que a XOR *a*' = 1, obtenemos que *U* valdrá siempre 111, y por lo tanto *V* valdrá siempre 000. Deducimos entonces que la función del circuito *M*1 es generar la combinación 000.
**c)** Analizamos los valores que toma *Y* en función de los valores que toma *X*.

Como hemos visto en el apartado *a)* analizando la salida del bloque *M*0, *s* vale 0 si *X* = 0, 1 o 2 y vale 1 siempre que *X* > 2. Por lo tanto, si *X* ≤ 2 la salida *Y* del circuito vale 0 (que es lo que vale *V* en todo momento) y si *X* > 2 vale *W*.
Analicemos ahora el valor de *W*:

**a)** Si *X* ≤ 2, no es necesario que lo analicemos porque sabemos que en este caso, *Y* vale 0.
**b)** Si *X* vale 3 o 4, *W* vale 3 y 4 respectivamente.
**c)** Si *X* ≥ 5, *W* vale 5.

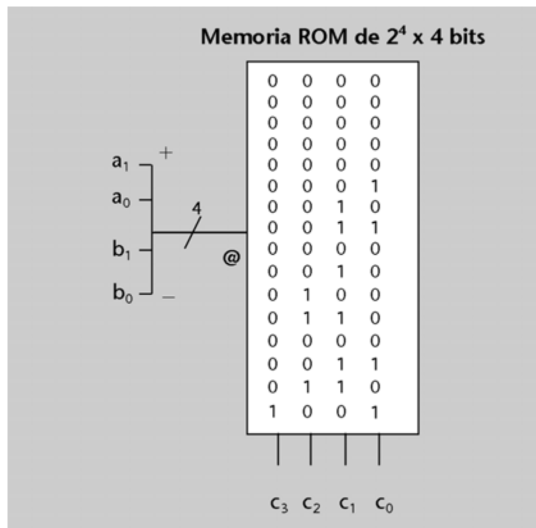
Juntándolo todo, obtenemos esta tabla de verdad:

| $x_2x_1x_0$ | $y_2y_1y_0$ |
|---|---|
| 000 | 000 |
| 001 | 000 |
| 010 | 000 |
| 011 | 011 |
| 100 | 100 |
| 101 | 101 |
| 110 | 101 |
| 111 | 101 |

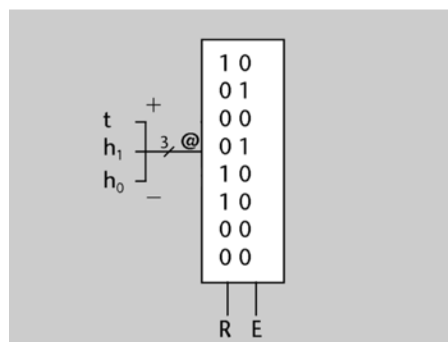
**47.** El circuito multiplicador de dos números naturales de dos bits tiene cuatro entradas ($a_1$ $a_0$ $b_1$ $b_0$) y cuatro salidas ($c_3$ $c_2$ $c_1$ $c_0$), tal como se explica en la actividad 24.

Cuatro variables pueden tomar $2^4$ = 16 combinaciones diferentes; por tanto, el tamaño de la ROM es de dieciséis palabras de 4 bits (1 para cada función de salida).

En la siguiente figura se muestra su contenido, que coincide con la parte derecha de la tabla de verdad de la actividad 24. La variable $a_1$ está conectada al bit de mayor peso del bus de direcciones y la variable $b_0$, al bit de menor peso (es muy importante indicar en el bus de direcciones a qué señal de entrada se conecta el bit de mayor peso para que el resultado sea correcto. Si los bits de entrada estuvieran en otro orden, la tabla de verdad sería diferente y, por tanto, también el contenido de la ROM).




**48.** Tal como se explica en la actividad 19, este circuito tiene tres señales de entrada y dos de salida y, por tanto, el tamaño de la memoria ROM es de $2^3 \times 2$ bits (ocho palabras de 2 bits). En el contenido de la memoria se han sustituido las x de la tabla de verdad por ceros, ya que el valor de cualquier bit en una memoria ROM debe estar definido. No obstante, las *x* se podrían haber sustituido por cualquier valor binario.




**49.** Conectaremos la entrada *X* a la entrada de direcciones de la memoria ROM, para que la salida nos dé la representación en complemento a 2 deseada. Por lo tanto, la entrada de direcciones será de 8 bits y la memoria tendrá 256 palabras de 8 bits cada una.

A la palabra de la dirección *i* le pondremos la representación en complemento a 2 del número que en signo y magnitud se codifica con la secuencia de bits correspondiente a la representación binaria del número *i*. De este modo, las direcciones 00000000..01111111 se accederán cuando el número *X* sea positivo, que en complemento a 2 se codifican igual que en signo y magnitud, de manera que en la palabra de cada una de estas direcciones habrá la misma combinación de bits que la de la dirección correspondiente. Las direcciones 10000000..11111111 se accederán cuando el número *X* sea negativo, de modo que en la palabra de cada una de estas direcciones habrá la representación en complemento a 2 del número –*A*, siendo *A* el número que codifican los 7 bits más bajos de la dirección.

La dirección 50 es, expresada en binario, la dirección 00110010. Si la leemos como un número expresado en signo y magnitud, vemos que es positivo y, por lo tanto, el contenido de esta dirección será 00110010.

La dirección 150 es, expresada en binario, la dirección 10010110. Si la leemos como un número expresado en signo y magnitud, vemos que es negativo. La magnitud del número es 0010110. Antes de cambiarle el signo, lo tenemos que expresar en 8 bits, ya que estamos buscando una representación en complemento a 2 y 8 bits: 00010110. Ahora cambiamos 1 por 0 y viceversa, sumamos 1 al resultado y obtenemos 11101010. Este será el contenido de la dirección 150.

La dirección 200 es, expresada en binario, la dirección 11001000. Si la leemos como un número expresado en signo y magnitud, vemos que es negativo. La magnitud del número es 1001000. De nuevo, antes de cambiarle el signo lo tenemos que expresar en 8 bits: 01001000. Ahora cambiamos 1 por 0 y viceversa y sumamos 1 al resultado, y obtenemos 10111000. Este será el contenido de la dirección 200.

**50.**
**a)** La tabla de verdad del circuito es la siguiente:

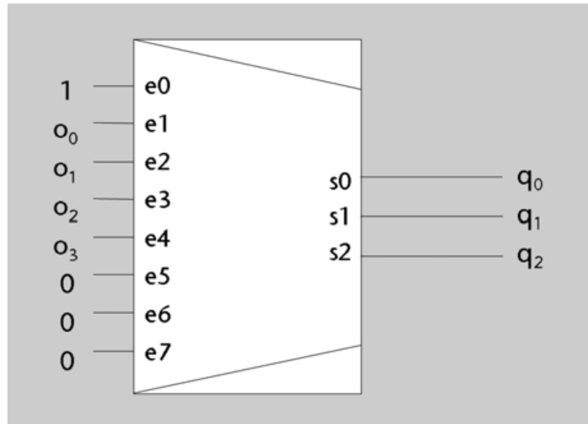
| $e_3$ $e_2$ $e_1$ $e_0$ | $o_3$ $o_2$ $o_1$ $o_0$ |
|---|---|
| 0 0 0 0 | 0 0 0 0 |
| 0 0 0 1 | 0 0 0 1 |
| 0 0 1 0 | 0 0 0 1 |
| 0 0 1 1 | 0 0 1 1 |
| 0 1 0 0 | 0 0 0 1 |
| 0 1 0 1 | 0 0 1 1 |
| 0 1 1 0 | 0 0 1 1 |
| 0 1 1 1 | 0 1 1 1 |
| 1 0 0 0 | 0 0 0 1 |
| 1 0 0 1 | 0 0 1 1 |
| 1 0 1 0 | 0 0 1 1 |
| 1 0 1 1 | 0 1 1 1 |
| 1 1 0 0 | 0 0 1 1 |
| 1 1 0 1 | 0 1 1 1 |
| 1 1 1 0 | 0 1 1 1 |
| 1 1 1 1 | 1 1 1 1 |

**b)** Las salidas del bloque ORDENAR tienen todos los unos a la derecha, tal como se muestra en la tabla siguiente (recordemos que CUANTOS = [$q_2$ $q_1$ $q_0$]:

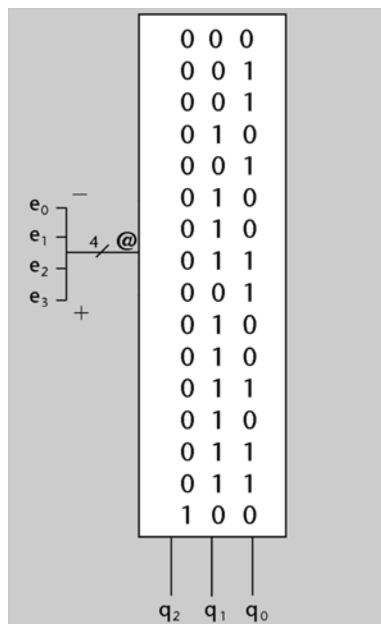
| $o_3$ $o_2$ $o_1$ $o_0$ | CUANTOS | $q_2$ $q_1$ $q_0$ |
|---|---|---|
| 0 0 0 0 | 0 (no hay 1 en la entrada) | 0 0 0 |
| 0 0 0 1 | 1 (hay un 1 en la entrada) | 0 0 1 |
| 0 0 1 1 | 2 (hay dos 1 en la entrada) | 0 1 0 |
| 0 1 1 1 | 3 (hay tres 1 en la entrada) | 0 1 1 |
| 1 1 1 1 | 4 (hay cuatro 1 en la entrada) | 1 0 0 |

Fijémonos en que la salida CUANTOS tiene que formar la codificación binaria de uno de los números 0, 1, 2, 3 ó 4. Para sintetizarla, por tanto, podemos usar un codificador 8-3 (son necesarios tres bits para codificar el cuatro en binario). Concretamente, tenemos que si el bit más alto de ORDEN que está en 1 es $o_i$, entonces CUANTOS tiene que valer $i + 1$. Por tanto, conectaremos la señal $o_i$ a la entrada $i + 1$ del codificador, tal como se muestra en la siguiente figura.

En la entrada 0 del codificador puede haber tanto un 1 como un 0 (en los dos casos, CUANTOS valdrá 0 si no hay ningún bit de ORDEN en 1).




**c)** El tamaño de la ROM debe ser $2^4 \times 3$ bits, y su contenido es el siguiente.




**51.** Primero, deduciremos la expresión algebraica de cada una de las funciones. Denominaremos $a_1$, $a_0$, $b_1$ y $b_0$ a los bits de los números de entrada. Para que los números sean iguales, lo deben ser bit a bit. Por tanto, se tiene que cumplir que $a_0 = b_0$ y $a_1 = b_1$.

La puerta XOR permite detectar la desigualdad entre 2 bits (recordad su tabla de verdad). Por tanto, una negación de su salida detectará la igualdad. Así pues, la expresión para la salida $A = B$ es la siguiente:

$$(a_1 \oplus b_1)'(a_0 \oplus b_0)'.$$

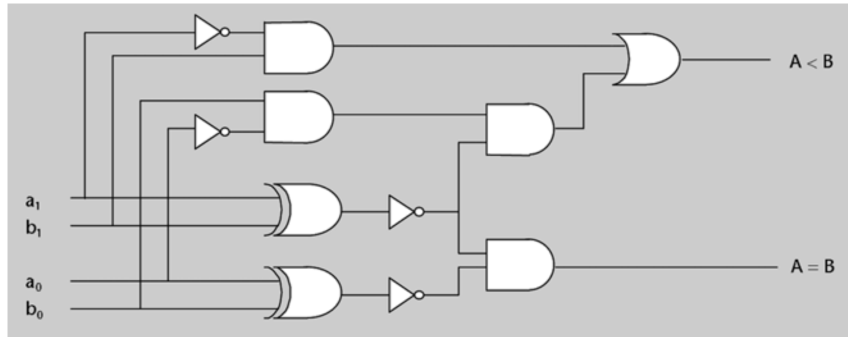
Para hacer la salida $A < B$ tendremos en cuenta lo siguiente:

– $A$ es menor que $B$ si lo es su bit de más peso: $b_1 = 1$ y $a_1 = 0$.
– Si los dos bits de más peso de $A$ y $B$ son iguales, entonces $A$ es menor que $B$ si lo es su bit de menos peso $a_1 = b_1$ y $b_0 = 1$ y $a_0 = 0$.

Por lo tanto, la expresión por la salida $A < B$ es la que mostramos a continuación:

$$b_1a_1' + (b_1 \oplus a_1)'b_0\, a_0'.$$

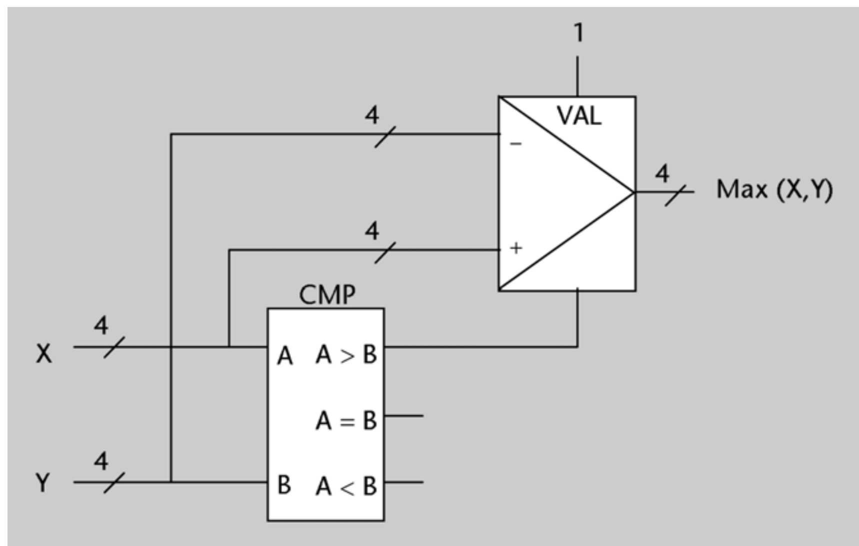
En la siguiente figura se encuentra la implementación de los dos circuitos.




**52.** Para deducir cuál de los dos números es mayor utilizaremos un comparador de cuatro bits. Conectaremos el número *X* a la entrada *A* y el número *Y* a la entrada *B* del comparador. La salida *A* > *B* de este comparador valdrá 1 si *X* > *Y*, y 0 en el caso contrario.

Esta salida controlará un multiplexor de buses que seleccionará entre *X* e *Y*. Así, cuando *A* > *B* vale 1 seleccionaremos el número *X*, y en cualquier otro caso (*X* < = *Y*) seleccionaremos el número *Y*.

La siguiente figura muestra este diseño.

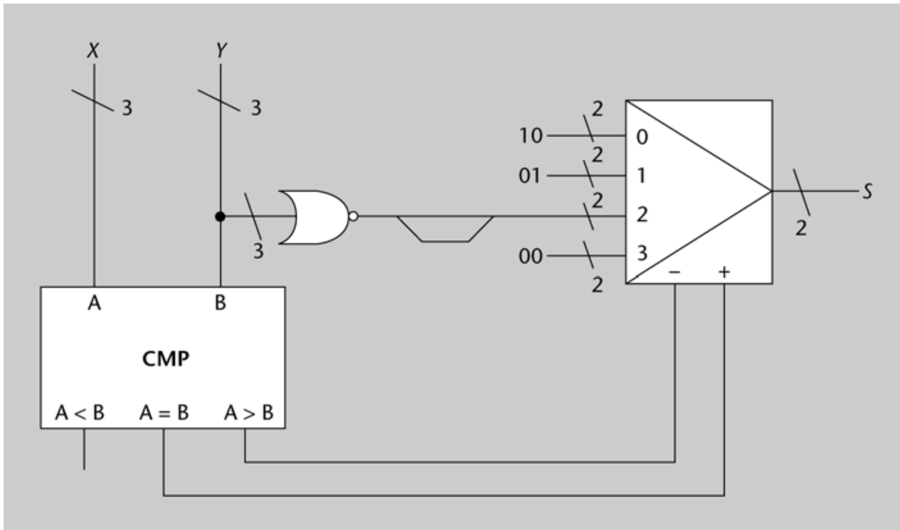



**53.** La salida *S* vale en todo momento alguno de cuatro valores. Una manera de implementarla es, por lo tanto, mediante un multiplexor 4-1.

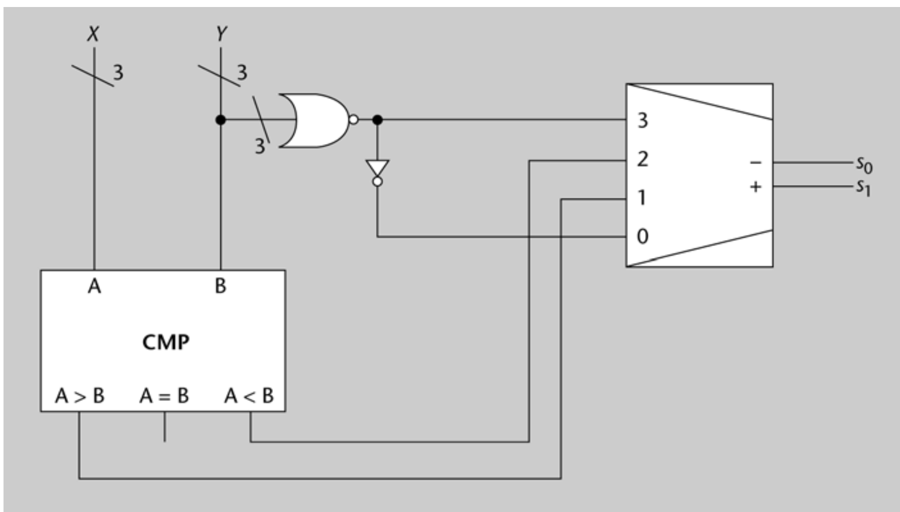
Cuál de estos valores deba elegirse en cada momento depende del resultado de la comparación entre *X* e *Y*, lo que nos lleva a usar un comparador, i, en caso de que *X* = *Y*, de si *X* (o *Y*) = 0; esto lo podemos saber usando una puerta NOR a cuyas entradas deberemos conectar los 3 bits de *X* o de *Y*.

Hay muchas posibilidades para controlar qué valor sale en cada momento del multiplexor a partir de las salidas del comparador y de la puerta NOR. Una es la que se muestra en la figura, según la cual:

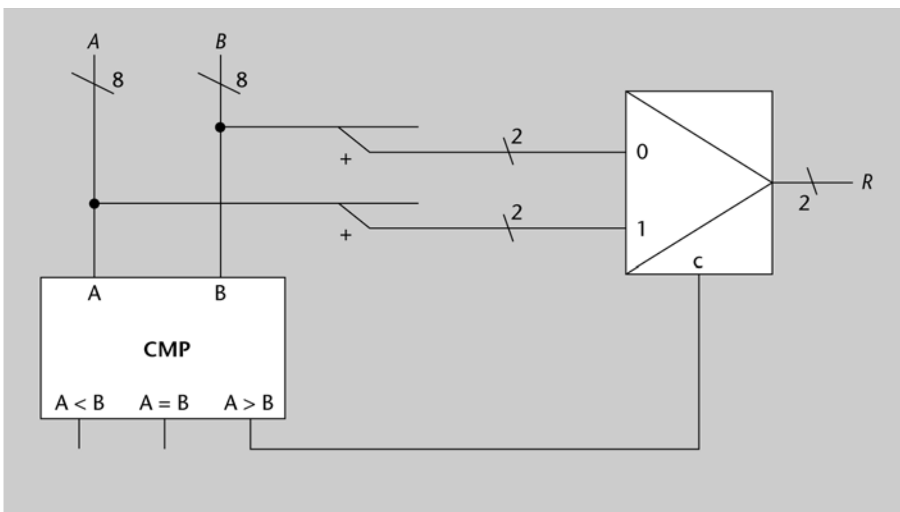
- La entrada de datos 0 pasará hacia la salida cuando *X* < *Y* (ni *X* = *Y* ni *X* > *Y*).
- La entrada de datos 1 pasará hacia la salida cuando *X* > *Y*.
- La entrada de datos 2 pasará hacia la salida cuando *X* = *Y*. En este caso, si la salida de la puerta NOR es 1 la salida *S* debe valer 11, y si es 0, *S* debe valer 00. Por lo tanto, podemos duplicar la salida de la puerta NOR para obtener el bus de dos bits que conectamos a la entrada de datos 2.
- La entrada de datos 3 no pasará nunca hacia la salida porque nunca se dará la combinación [1 1] en las entradas de control. Aquí podemos conectar por lo tanto cualquier cosa, por ejemplo 00.




También podemos diseñar el circuito sin multiplexor, generando los valores que puede tomar la salida con un codificador 4-2, ya que los valores que puede tomar son las cuatro combinaciones que se pueden hacer con dos bits. Observemos que si $X > Y$ o $X < Y$ la entrada 0 del codificador estará a 1, pero en la salida no se generará la combinación [0 0] porque habrá otra entrada del codificador también a 1.

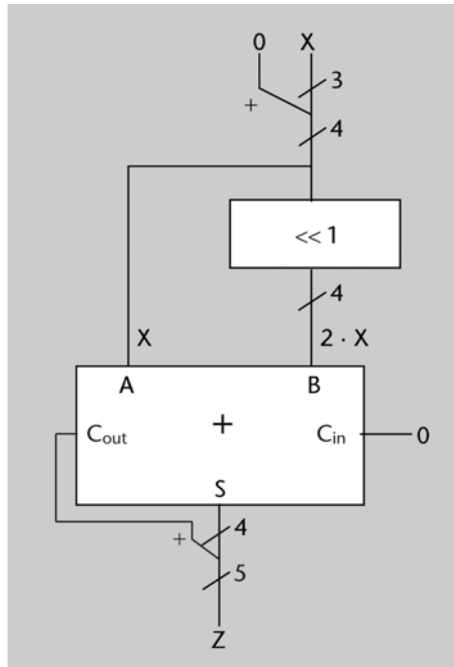



**54.** Para saber cuál de los dos hornos está más caliente, usaremos un comparador. Observemos que el valor de $R$ coincide en todo momento con los 2 bits más altos de la temperatura del horno que está más caliente (o igual), de manera que podemos generar $R$ a partir de estos bits. Un multiplexor selecciona si se deben tomar de la temperatura $A$ o de la $B$.



**55.** Para implementar esta función será suficiente con sumar $X + 2 * X$ ($X$ desplazando un bit a la izquierda). Observad que el resultado final debe tener 5 bits para que no se produzca desbordamiento.

La siguiente figura muestra este diseño.

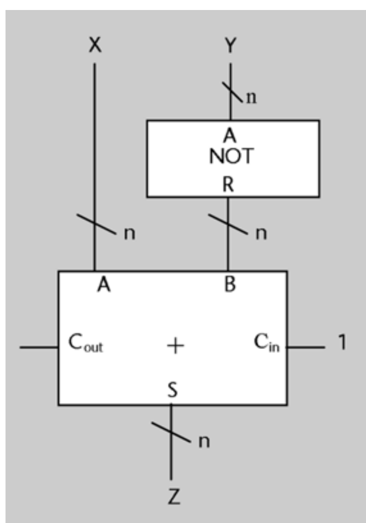



**56.** Para realizar esta operación aprovecharemos la siguiente igualdad, válida cuando los números están representados en complemento a 2:

$$Z = X - Y = X + (-Y) = X + Y' + 1.$$

Por tanto, para implementar el circuito necesitamos un bloque NOT de $n$ bits para obtener $Y'$, y un sumador de $n$ bits para hacer la suma $X + Y'$. El 1 que aún queda por sumar se puede conectar a la entrada de acarreo del sumador.

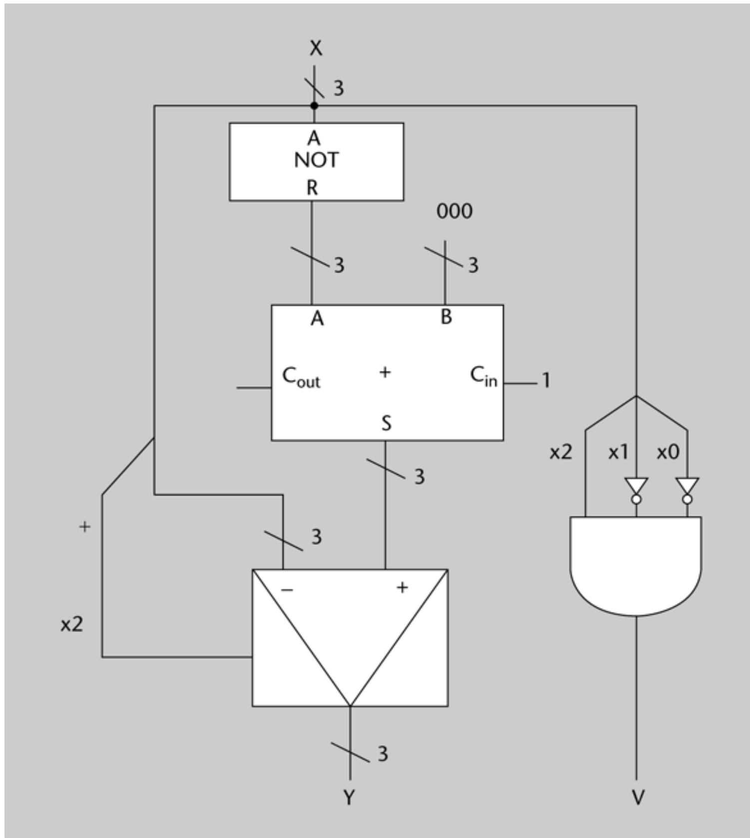
La siguiente figura muestra el diseño propuesto.




**57.** En caso de que el número sea positivo, la salida tiene que ser igual a la entrada, y en caso de que sea negativo es preciso cambiar el signo de la entrada; esto lo conseguimos negando

todos los bits y sumando 1 al resultado. Para seleccionar una opción u otra, utilizamos el bit de más peso de la entrada.

Sólo se puede producir desbordamiento en el caso de que la entrada valga –4, ya que en este caso el valor de la salida (4) no es representable con 3 bits en complemento a 2. Por lo tanto, $V = x_2x_1'x_0'$.




**58.**

**a)** La tabla de verdad es la siguiente (aquí escribimos también la interpretación de los números en complemento a 2 para hacerla más comprensible). Para calcular el resultado de la multiplicación pasamos las entradas a decimal como en la actividad 24.

| *A* | *B* | $a_1$ | $a_0$ | $b_1$ | $b_0$ | $m_3$ | $m_2$ | $m_1$ | $m_0$ | *M = A*B* |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | –2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | –1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | –2 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | –2 |
| 1 | –1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | –1 |
| –2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| –2 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | –2 |
| –2 | –2 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 4 |
| –2 | –1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 2 |
| –1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| –1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | –1 |
| –1 | –2 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 2 |
| –1 | –1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

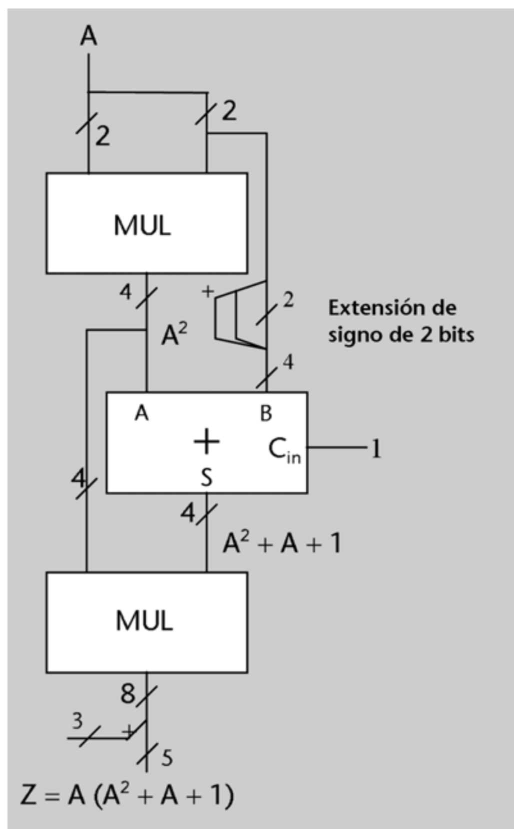
**b)** Para obtener la solución que se describe en el enunciado, con un sumador y dos multiplicadores se debe utilizar la siguiente igualdad:

$$A^4 + A^3 + A^2 = A^2\ (A^2 + A + 1).$$

A continuación haremos un estudio del rango que pueden tener cada una de las expresiones para decidir el número de bits de cada bloque..

| $a_1\ a_0$ | $A$ | $A^2$ | $A^2 + A + 1$ | $A^2 \cdot (A^2 + A + 1) = A^4 + A^3 + A^2$ |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 1 | 1 | 3 | 3 |
| 10 | –2 | 4 | 3 | 12 |
| 11 | –1 | 1 | 1 | 1 |
| se necesitan: | | 4 bits | 3 bits | 5 bits |

$A^2$ se obtendrá mediante un bloque MUL de 2 bits, conectando A a las dos entradas. $A^2$, pues, tendrá 4 bits (el doble de bits que las entradas).
$A^2 + A + 1$ se obtendrá mediante un sumador, conectando el 1 a la entrada $C_{in}$. Aunque si nos fijamos en el rango del resultado de esta suma con 3 bits sería suficiente, el sumador debe ser de 4 bits porque $A^2$ tiene 4 bits. Por tanto, será necesario ampliar a 4 bits la entrada $A$ (haciendo una extensión de signo) para conectarla a la otra entrada del sumador, tal como se muestra en la figura.
Otro bloque MUL de 4 bits calculará $A^2\ (A^2 + A + 1)$. La salida del multiplicador será, pues, de 8 bits, aunque el rango del resultado sólo necesita 5. Éste está formado, por tanto, por los 5 bits de menor peso de la salida de este segundo bloque MUL.



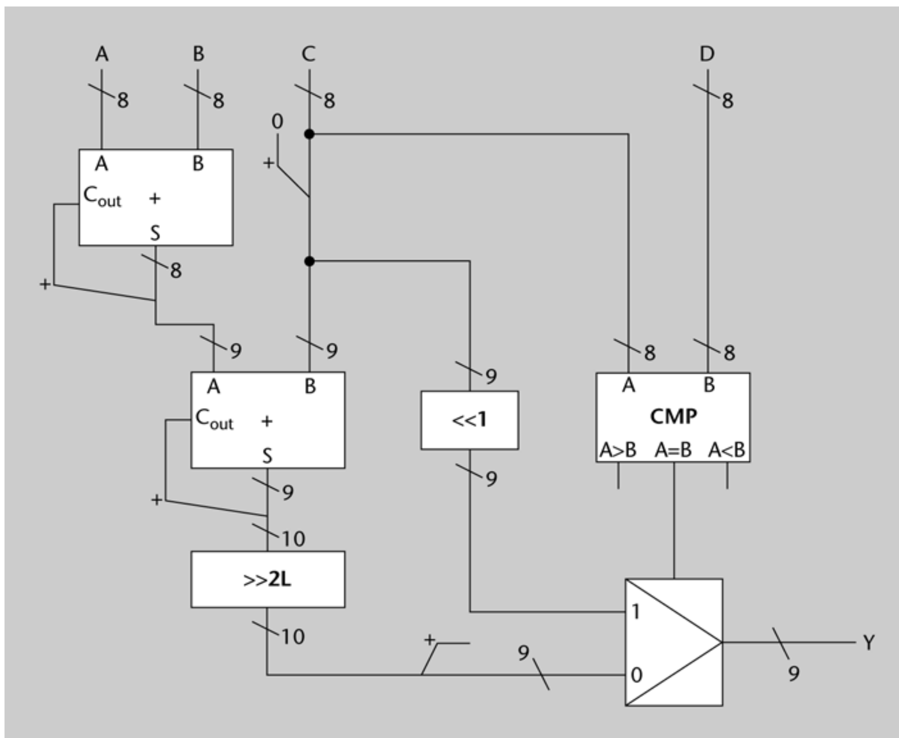

**59.** Puesto que $A$, $B$, $C$ y $D$ codifican números naturales con 8 bits, pueden valer entre 0 y 255.

Multiplicaremos $C$ por 2 mediante un desplazador a la izquierda de 9 bits, porque el resultado será un valor entre 0 y 510. Por lo tanto, extendemos $C$ a 9 bits antes de conectarlo a la entrada del desplazador.

Por otro lado, para obtener la suma $A + B + C$ calcularemos primero $A + B$ con un sumador de 8 bits. El resultado valdrá entre 0 y 510, y por lo tanto se tiene que representar con 9 bits: los alcanzaremos agregando al resultado el $C_{out}$ del sumador. Después sumamos a este resultado el valor de $C$ extendido a 9 bits, y obtendremos un valor entre 0 y 765, que se debe representar con 10 bits. De nuevo los obtendremos agregando al resultado el $C_{out}$ del sumador.

Para dividir la suma entre 4 se tiene que desplazar 2 bits a la derecha, lo cual podemos hacer con un desplazador de 10 bits. Puesto que estamos trabajando con números naturales, el desplazador debe ser lógico. El resultado de la división es un valor entre 0 y 191, que requiere 8 bits para ser representado. Sin embargo, puesto que $Y$ debe tener 9 bits para representar cualquiera de los dos posibles resultados (ya hemos visto que $2 \times C$ requiere 9 bits), descartaremos sólo uno de los bits de salida del desplazador.

Finalmente, comparamos $C$ y $D$ para saber cuál de los dos cálculos tenemos que dar como resultado. Elegimos entre los mismos mediante un multiplexor.

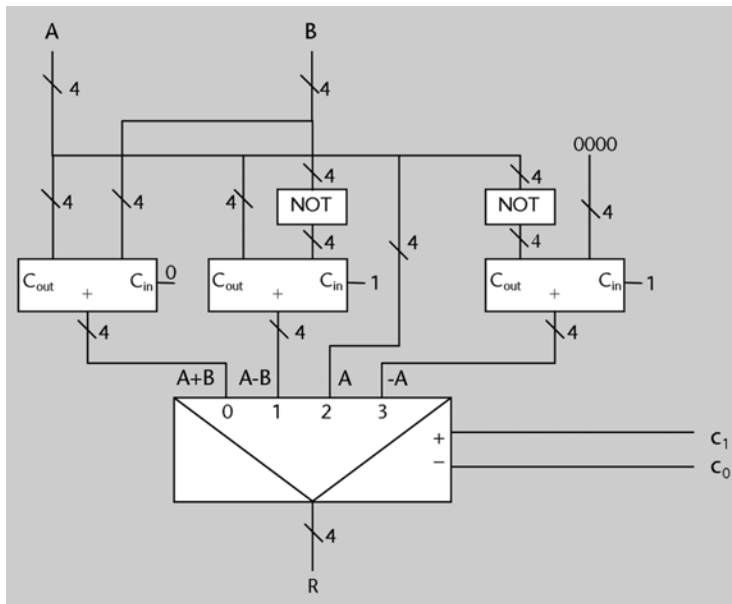



**60.**

**a)** Debemos diseñar una UAL que pueda realizar cuatro operaciones. Algunas de éstas se pueden hacer directamente usando bloques que ya hemos estudiado en la teoría de este módulo; sin embargo, otras requieren la utilización de circuitos que combinen bloques con puertas. Analizaremos una por una las operaciones para decidir cómo haremos su implementación. Tal como hemos visto en la teoría, una vez se han implementado los circuitos que hacen todas las operaciones, seleccionaremos la operación que se debe realizar mediante un multiplexor de buses de 4 bits de cuatro entradas.

- $R = A + B$ Esta suma se puede hacer directamente con un sumador de 4 bits.
- $R = A - B$ Tal como se ha visto en la actividad 56, $A - B = A + B' + 1$. Por tanto, para implementar esta resta harán falta un sumador y un bloque NOT de 4 bits.
- $R = A$ La entrada $A$, de 4 bits, estará conectada directamente a una de las entradas del multiplexor.
- $R = -A$ Como hemos hecho en el caso de la resta, podemos usar la igualdad $-A = A' + 1$ y, por tanto, para implementar esta operación necesitaremos un sumador y un bloque NOT de 4 bits.

En la siguiente figura se puede ver la implementación descrita.




**b)** Analizamos cómo hay que calcular cada uno de estos bits. Fijémonos en que la operación $R = A$ nunca generará un desbordamiento.

- *Vb*: para estudiar el valor de este bit se deben interpretar las entradas (y, por tanto, también la salida) como números naturales codificados en binario.

– Operación $R = A + B$: en este caso el desbordamiento es el acarreo generado en el último bit.
– Operación $R = A - B$: se producirá desbordamiento cuando el resultado sea negativo (no se podrá codificar en binario). Teniendo en cuenta que para restar sumamos el complementario más 1, el resultado es negativo cuando no se produce acarreo (podéis verificar esta afirmación probando unos cuantos ejemplos).
– Operación $R = -A$: se produce desbordamiento siempre, porque el resultado siempre es un número negativo.

La siguiente tabla resume el valor de *Vb*:

| $c_1$ | $c_0$ | *Vb* |
|---|---|---|
| 0 | 0 | $C_{out}$ |
| 0 | 1 | $C_{out}'$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- *V*: para estudiar el valor de este bit es necesario que nos pongamos en el caso de interpretar las entradas y, por tanto, también la salida, como números enteros codificados en complemento a 2.

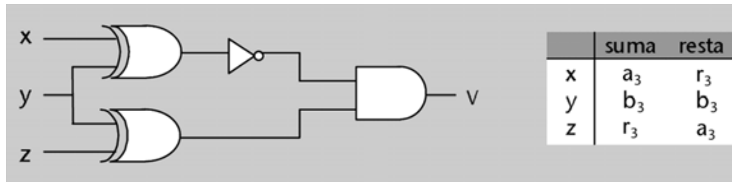
– Operación $R = A + B$: se produce desbordamiento si los dos sumandos son del mismo signo y el resultado es de signo contrario. El signo de los operandos y del resultado viene determinado por su bit de mayor peso. Como en la actividad 51, podemos usar la operación XOR para detectar igualdad o desigualdad entre estos bits y, de este modo, obtener la siguiente expresión:

$$V = (a_3 \oplus b_3)' \, (b_3 \oplus r_3).$$

– Operación $R = A - B$: se produce desbordamiento cuando los dos operandos son de signo contrario y el resultado es del mismo signo que B. Por tanto,

$$V = (a_3 \oplus b_3) \, (b_3 \oplus r_3)'.$$

Como se puede observar en las dos expresiones anteriores, se puede utilizar el mismo circuito para calcular el desbordamiento para $A + B$ y para $A - B$, aunque las entradas se deben conectar de manera diferente. Este circuito tiene tres entradas, *x*, *y* y *z*, y calcula la expresión $s = (x \oplus y)' \cdot (y \oplus z)$. Lo hemos llamado *detector V*, y su diseño interno se presenta a continuación.




– Operación $R = -A$. Se produce desbordamiento sólo en el caso $A = -8$ y, por tanto:

$$V = a_3\ a_2'\ a_1'\ a_0'.$$

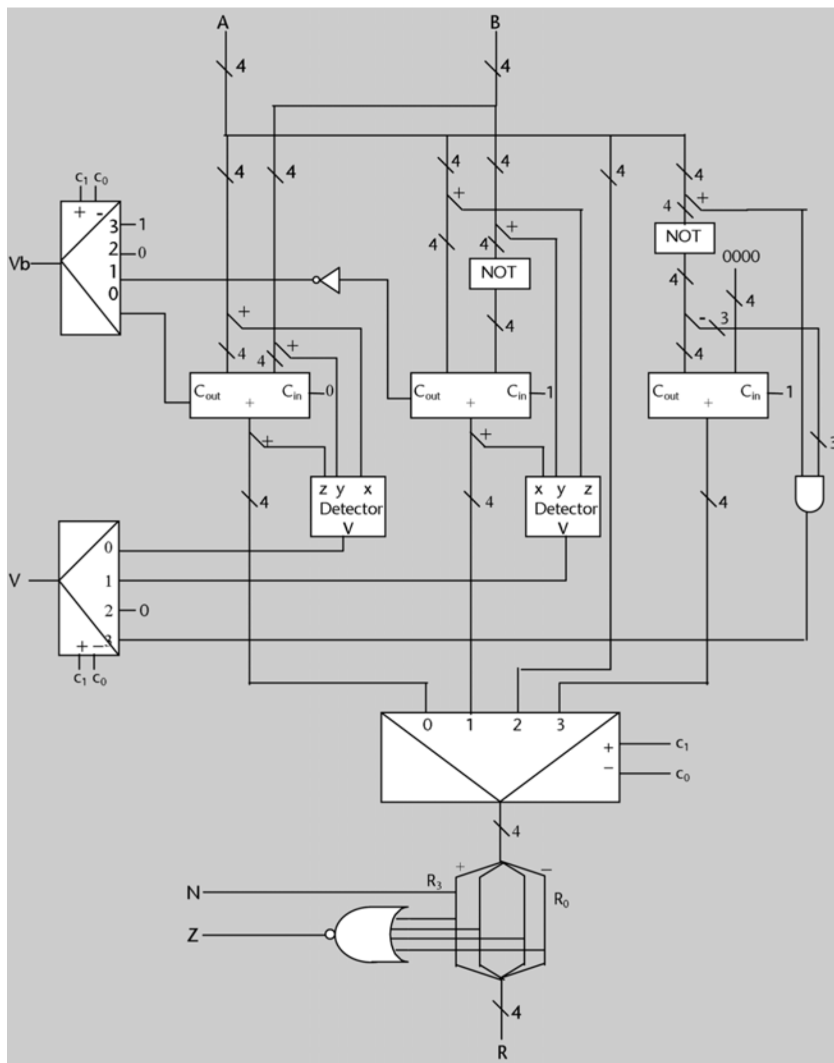
La siguiente tabla resume el valor del bit $V$:

| $c_1$ | $c_0$ | $V$ |
|---|---|---|
| 0 | 0 | $(a_3 \oplus b_3)'(b_3 \oplus r_3)$ |
| 0 | 1 | $(r_3 \oplus b_3)'(b_3 \oplus a_3)$ |
| 1 | 0 | 0 |
| 1 | 1 | $a_3\ a_2'\ a_1'\ a_0'$ |

Para implementar los dos bits de desbordamiento utilizaremos dos mutiplexores, los dos controlados por los bits $c_1$ y $c_0$. Los multiplexores seleccionarán en cada momento el valor correcto de $V$ y $Vb$ según la operación que se realice.

- $N$: tal como se ha visto en la teoría, el signo del resultado es el bit de más peso.
- $Z$: tal como se ha visto en la teoría, para calcular si el resultado es 0 se hace con una puerta NOR de cuatro entradas.

A continuación, se muestra el circuito.



## Ejercicios de autoevaluación

**1.** Debemos demostrar que $(xyz)' = x' + y' + z'$. En primer lugar aplicamos la propiedad asociativa y, después, el teorema de De Morgan para dos variables, dos veces:

$$(xyz)' = ((xy)z)' = (xy)' + z' = x' + y' + z'.$$

Haremos lo mismo para cuatro variables, habiendo demostrado ya que se cumple para tres variables:

$$(xyzw)' = ((xyz)w)' = (xyz)' + w' = x' + y' + z' + w'.$$

**2.**
$F = (((w'\,y)' + wx)\,z + x')\,y' = ((w'' + y' + wx)\,z + x')\,y' = ((w + y' + wx)\,z + x')\,y' = ((w + y') + z + x')\,y' =$
$= (wz + y'z + x')\,y' = wzy' + y'y'z + x'y' = wzy' + y'z + x'y'.$

**3.** Primero haremos la tabla de verdad. Después implementaremos el circuito a dos niveles.

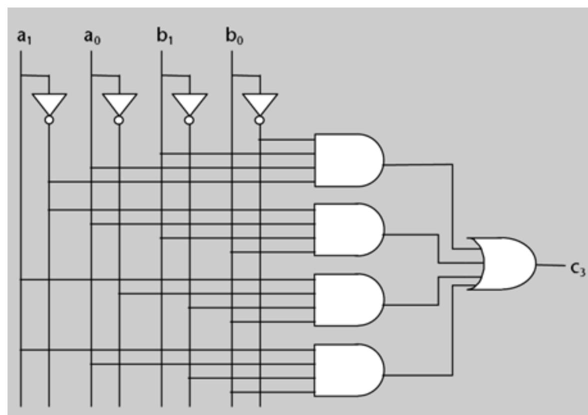
El rango de valores que puede tomar un número entero de dos bits representado en complemento a dos es [–2,1]. Por tanto, el rango de la multiplicación de dos números será [–2..4]. Para representar el 4 en complemento a 2 necesitamos 4 bits. Por tanto, la salida tendrá 4 bits.

Llamaremos $[a_1\,a_0]$ y $[b_1\,b_0]$ respectivamente a los bits de los dos números de entrada, y $[c_3\,c_2\,c_1\,c_0]$ a los bits de su multiplicación. La tabla de verdad de este multiplicador es la siguiente:

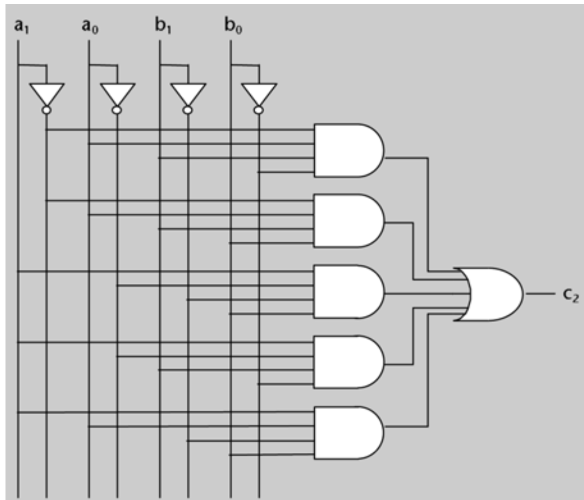
| $a_1$ | $a_0$ | $b_1$ | $b_0$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

A continuación, se muestran los circuitos a dos niveles correspondientes a las cuatro funciones de salida.

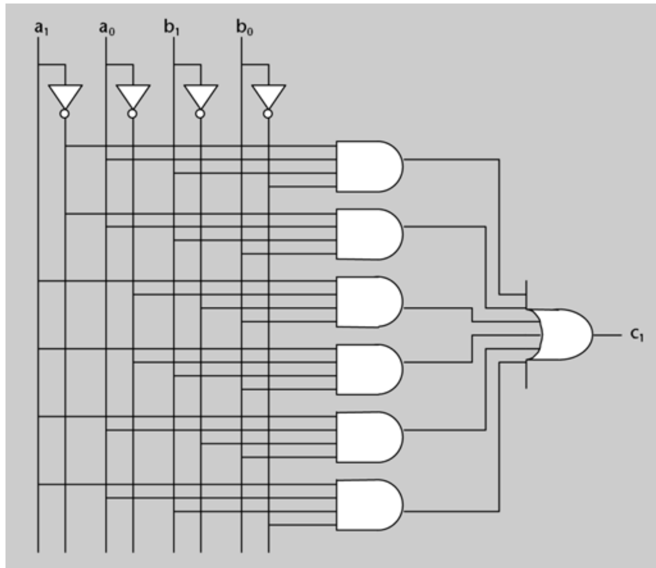
$c_3 = a_1'\,a_0\,b_1\,b_0' + a_1'\,a_0\,b_1\,b_0 + a_1\,a_0'\,b_1'\,b_0 + a_1\,a_0\,b_1'\,b_0.$



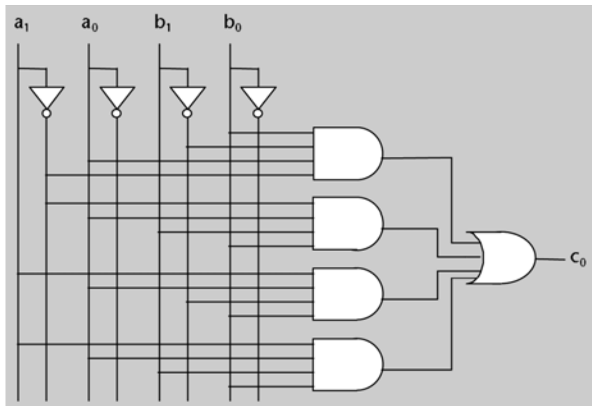
$c_2 = a_1' \, a_0 \, b_1 \, b_0' + a_1' \, a_0 \, b_1 \, b_0 + a_1 \, a_0' \, b_1' \, b_0 + a_1 \, a_0' \, b_1 \, b_0' + a_1 \, a_0 \, b_1' \, b_0$.




$c_1 = a_1' \, a_0 \, b_1 \, b_0' + a_1' \, a_0 \, b_1 \, b_0 + a_1 \, a_0' \, b_1' \, b_0 + a_1 \, a_0' \, b_1 \, b_0 + a_1 \, a_0 \, b_1' \, b_0 + a_1 \, a_0 \, b_1 \, b_0'$.



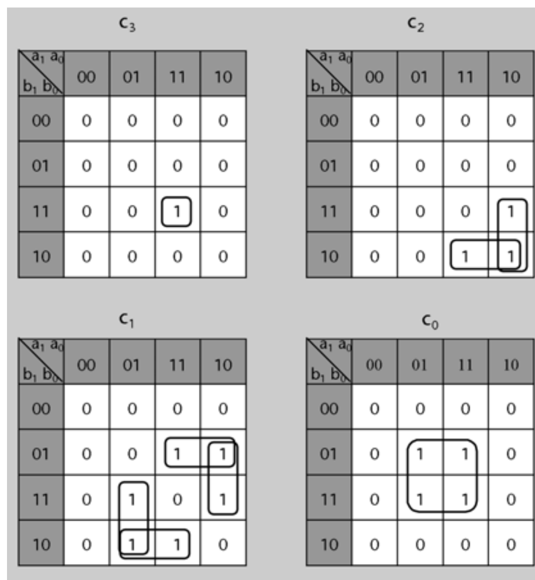

$c_0 = a_1' \, a_0 \, b_1' \, b_0 + a_1' \, a_0 \, b_1 \, b_0 + a_1 \, a_0 \, b_1' \, b_0 + a_1 \, a_0 \, b_1 \, b_0$.



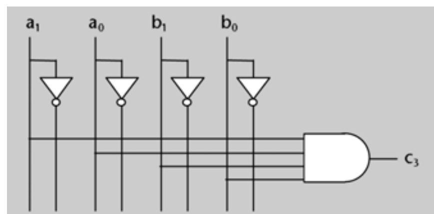
**4.** Tenemos la siguiente tabla de verdad:

| $a_1$ | $a_0$ | $b_1$ | $b_0$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

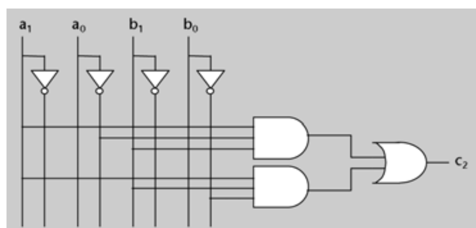
A continuación, simplificaremos por Kanaugh cada una de las funciones de salida y, después, implementaremos el circuito.



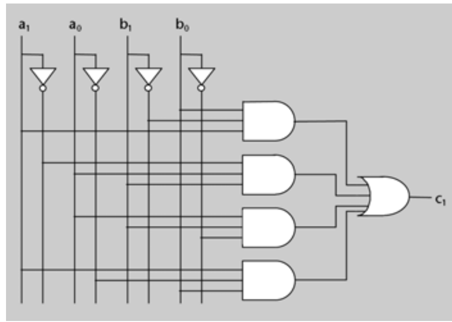

$c_3 = a_1a_0b_1b_0$.



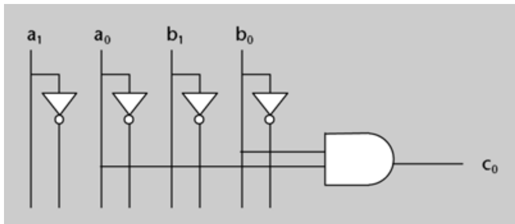

$c_2 = a_1a_0'b_1 + a_1b_1b_0'$.



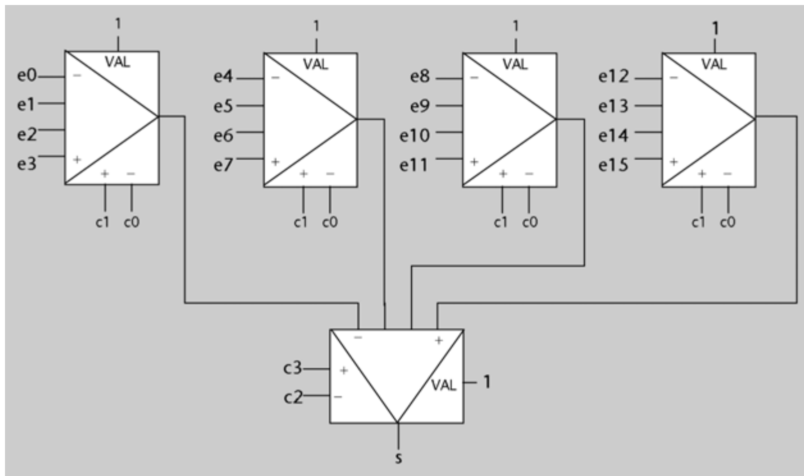
$c_1 = a_1b_1'b_0 + a_1'a_0b_1 + a_0b_1b_0' + a_1a_0'b_0$.



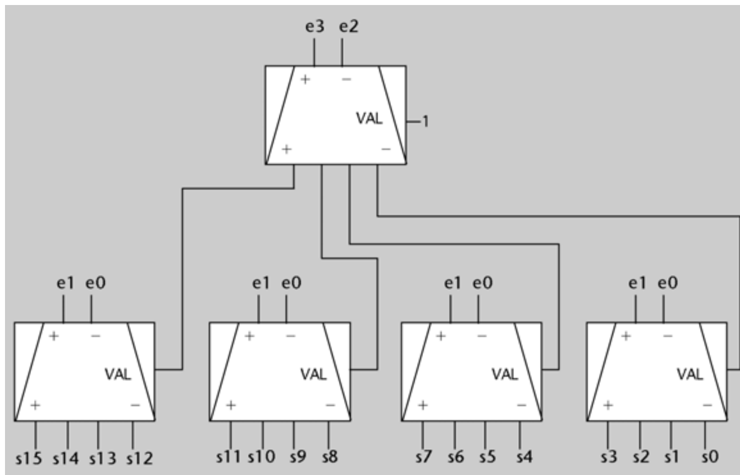

$c_0 = a_0b_0$.




**5.**



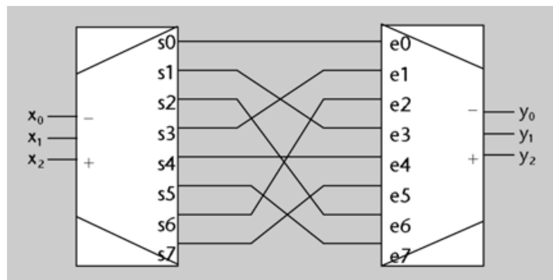

**6.**



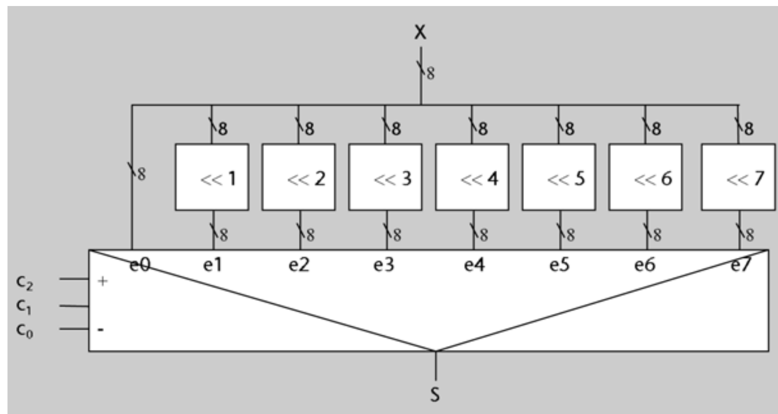
**7.** La tabla de verdad es la siguiente::

| $X$ | $x_2$ | $x_1$ | $x_0$ | $y_2$ | $y_1$ | $y_0$ | $Y = 3X \bmod 8$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 3 |
| 2 | 0 | 1 | 0 | 1 | 1 | 0 | 6 |
| 3 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 | 0 | 0 | 4 |
| 5 | 1 | 0 | 1 | 1 | 1 | 1 | 7 |
| 6 | 1 | 1 | 0 | 0 | 1 | 0 | 2 |
| 7 | 1 | 1 | 1 | 1 | 0 | 1 | 5 |

A continuación se muestra su diseño.




**8.**
**a)**




**b)** Teniendo en cuenta que una operación de desplazamiento a la izquierda de un bit es equivalente a multiplicar por 2, si desplazamos $X$ $n$ bits, esta operación equivale a $S = X \cdot 2n$.

**c)** Si $C = 010$, el desplazamiento corresponde a la operación de multiplicar por 4. Si $X$ es natural, el rango representable con 8 bits es [0..255]. Por tanto, se produce desbordamiento cuando $X \geq 256 / 4 = 64$.

Si $X$ es entero, el rango de número representables es [–128..127]. Por tanto, se produce desbordamiento cuando $X \geq 128 / 4 = 32$ o bien cuando $X < -128 / 4 = -32$.

**d)** Si $X$ es un número natural, los bits de más peso que se añaden al número cuando éste se desplaza a la derecha deben valer 0. Por tanto, se debe usar un desplazador lógico.

Si $X$ es un número entero, los bits de más peso que se añaden al número cuando éste se desplaza a la derecha deben valer lo mismo que el bit de mayor peso del número para conservar el signo. Por tanto, se debe usar un desplazador aritmético.

**e)** Un desplazador a la derecha no computa la función $X / 2^n$, sino la función $\lfloor X / 2^n \rfloor$. Por tanto, si se usa para calcular $X / 2^n$, el resultado no será exacto si alguno de los bits que se pierden al desplazar a la derecha vale 1. Dicho de otra manera, si el desplazamiento es de $k$ bits ($C = k$), el resultado es inexacto cuando tenemos lo siguiente:

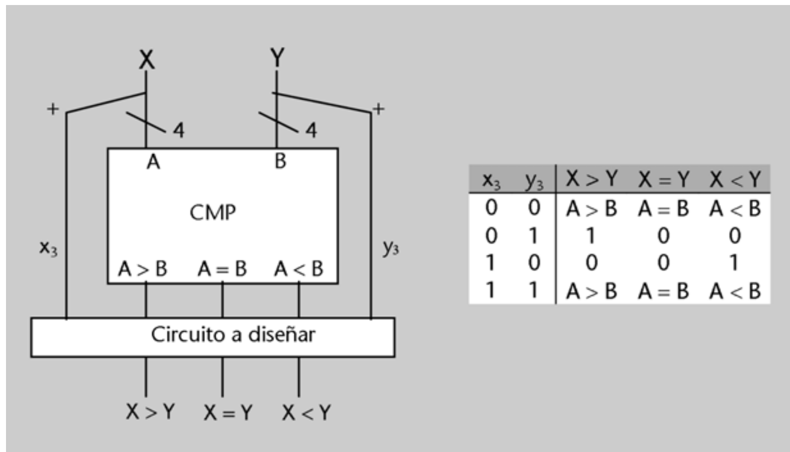
$$X \bmod 2^k \neq 0.$$

**9.** El resultado de la comparación depende de los signos de los dos números (que vienen determinados por su bit de más peso: 1 para los negativos, 0 para los positivos o cero).

Si los dos números son de signos diferentes, entonces el mayor es positivo.

Cuando los dos números son del mismo signo, entonces los podemos comparar usando un comparador de números naturales. Fijémonos en que el resultado será correcto también en el caso de que los dos números sean negativos, porque si $X > Y$, interpretando $X$ e $Y$ en complemento a dos, entonces también se cumple que $X > Y$ si los interpretamos en binario. Por ejemplo, si comparamos $X = -1$ (1111) con $Y = -2$ (1110) el comparador sacaría un 1 para la salida $X > Y$.

A continuación se muestra un primer esquema del circuito y la tabla de verdad que describe el comportamiento de la parte del circuito que queda por implementar, de acuerdo con el signo de los dos números.

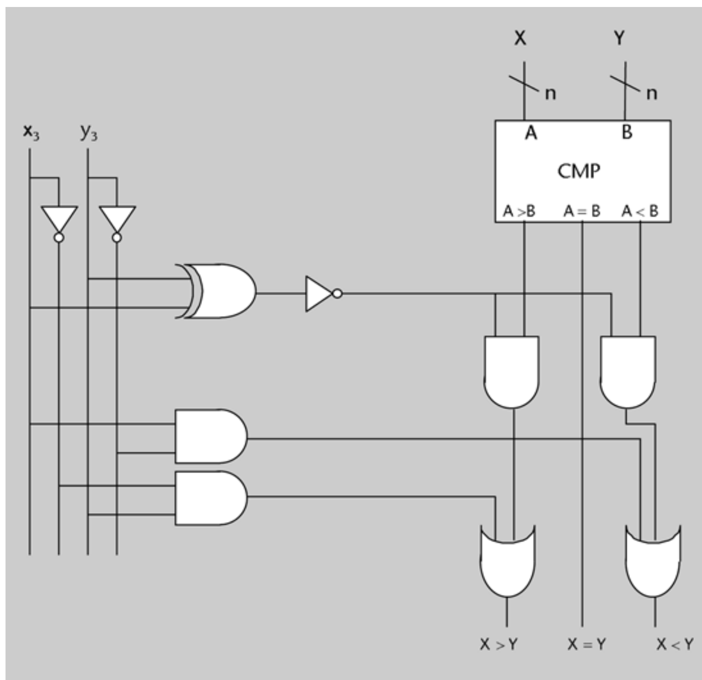



| $x_3$ | $y_3$ | X > Y | X = Y | X < Y |
|---|---|---|---|---|
| 0 | 0 | A > B | A = B | A < B |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | A > B | A = B | A < B |

De estas tablas se deducen las siguientes igualdades:

$[X > Y] = x_3'y_3 + (x_3 \oplus y_3)' \cdot [A > B]$,
$[X < Y] = x_3y_3' + (x_3 \oplus y_3)' \cdot [A < B]$,
$[X = Y] = [A = B]$.

A continuación se muestra el circuito completo:



**10. a)**La tabla de verdad de un *full adder* es la siguiente:

| $a_i$ | $b_i$ | $c_i$ | $c_{i+1}$ | $s_i$ |
|---|---|---|---|---|
| 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 |

La expresión en suma de productos de $c_{i+1}$ es la siguiente:

$$c_{i+1} = a_i'\ b_i\ c_i + a_i\ b_i'\ c_i + a_i\ b_i\ c_i' + a_i\ b_i\ c_i$$

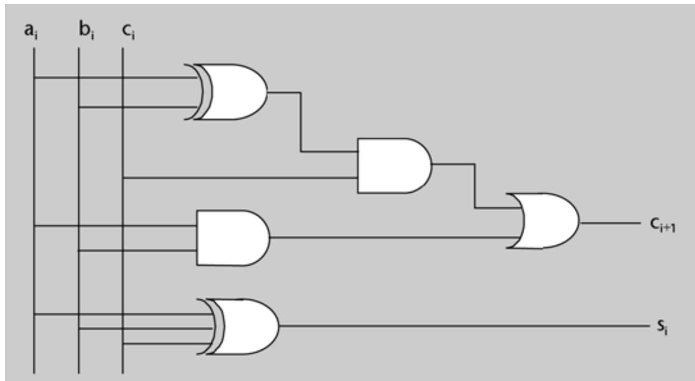
Podemos sacar factor común en los dos primeros y los dos últimos términos, y obtenemos lo siguiente:

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

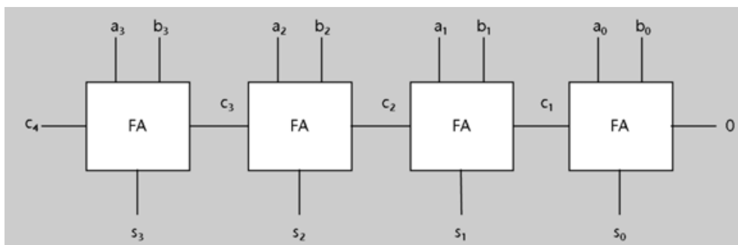
Por lo que se refiere a $s_i$, vemos que cuando $a_i = 0$ vale $(b_i \oplus c_i)$ y cuando $a_i = 1$ vale $(b_i \oplus c_i)'$. Esto se puede expresar de la siguiente manera:

$$s_i = a_i'\ (b_i \oplus c_i) + a_i\ (b_i \oplus c_i)' = a_i \oplus b_i \oplus c_i$$

El circuito interno de un *full adder*, que implementa estas expresiones, se muestra en la siguiente figura:




**b)** A continuación, se muestra cómo se encadenan cuatro *full adders* para conseguir un sumador de números de 4 bits. Como se puede ver en la figura, en el bit de acarreo de entrada, $c_0$, se debe conectar un 0 para que la suma sea correcta.




**11.** Se trata de diseñar un circuito que haga lo siguiente:

Si $s'/r = 0$, entonces $R = A + B$.
Si $s'/r = 1$, entonces $R = A - B = A + B' + 1$.

Podemos utilizar un sumador de números de cuatro bits para hacer este circuito.

– Conectaremos el número A a una entrada.

– En la otra entrada conectaremos el número *B* o el número *B'*, dependiendo de si la señal *s'/r* vale 0 o 1 respectivamente. Recordemos esta propiedad de la función XOR:

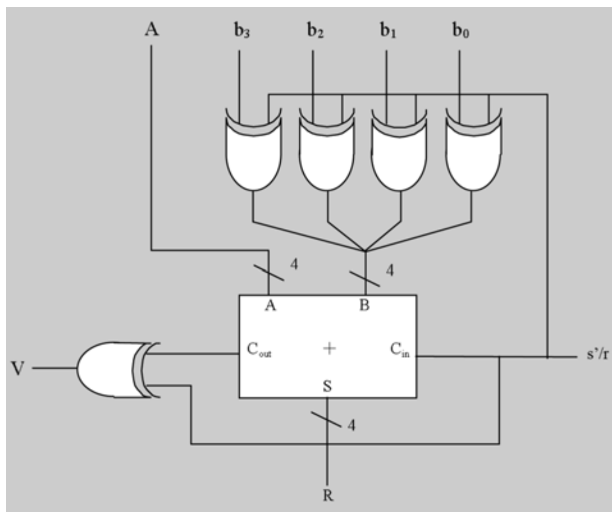
$$0 \oplus x = x,\ 1 \oplus x = x'$$

Por tanto, para obtener *B* o *B'* según *s'/r*, podemos hacer una XOR de *s'/r* con cada uno de los bits de *B*.

– Conectaremos un 0 a la entrada de acarreo si *s'/r* = 0, o bien un 1 si *s'/r* = 1. Por tanto, aquí podemos conectar directamente la señal *s'/r*.

Otra opción para seleccionar *B* o *B'* con *s'/r* sería utilizando un multiplexor.

El circuito sumador/restador que se describe en los párrafos anteriores se muestra a continuación.




Cuando se realiza una suma, la salida *V* coincide con el último bit de acarreo.

Cuando se hace la resta mediante la suma del complementario más 1, el acarreo del último bit es siempre la negación de lo que obtendríamos si se hubiera hecho directamente la resta.

Por tanto, cuando *s'/r* = 0 se cumple que $V = C_{out}$ y cuando *s'/r* = 1 se cumple que $V = C_{out}'$. Así pues, si utilizamos la misma propiedad de la función XOR que hemos utilizado antes, obtenemos lo siguiente:

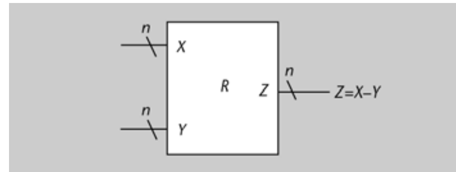
$$V = s'/r \oplus C_{out}$$

**12.** Para saber hacia qué planta se tienen que mover los ascensores (es decir, qué valor tenemos que dar a la señal *destino*), necesitamos codificar en binario (número natural) la planta desde la que se ha hecho la llamada. Esto lo podemos conseguir mediante un codificador de 8 entradas y 3 salidas, con $b_i$ conectado a la entrada $e_i$, las entradas $e_7$ y $e_6$ conectadas a cero y con los 3 bits de salida juntos formando el bus destino.

Para saber qué ascensor está más cerca de la planta de destino, podemos restar la planta donde está cada uno de los ascensores (*plantaA* y *plantaB*) de la planta *destino*. Para hacer la resta, podemos utilizar un sumador y aplicar la igualdad siguiente:

$$X - Y = X + (-Y) = X + Y' + 1$$

tal y como se hace en la actividad 56. Puesto que $X$ e $Y$ se deben representar en complemento a 2, añadiremos un 0 como bit de más peso en todos los operandos de las restas. Representamos el circuito que se diseña en esta actividad mediante el bloque $R$, de la manera siguiente:




La resta de dos valores en el rango [0, 5] dará un valor en el rango [–5, 5]. Tenemos suficiente con 4 bits para codificar este rango en complemento a 2, y por lo tanto $n$ puede ser 4.

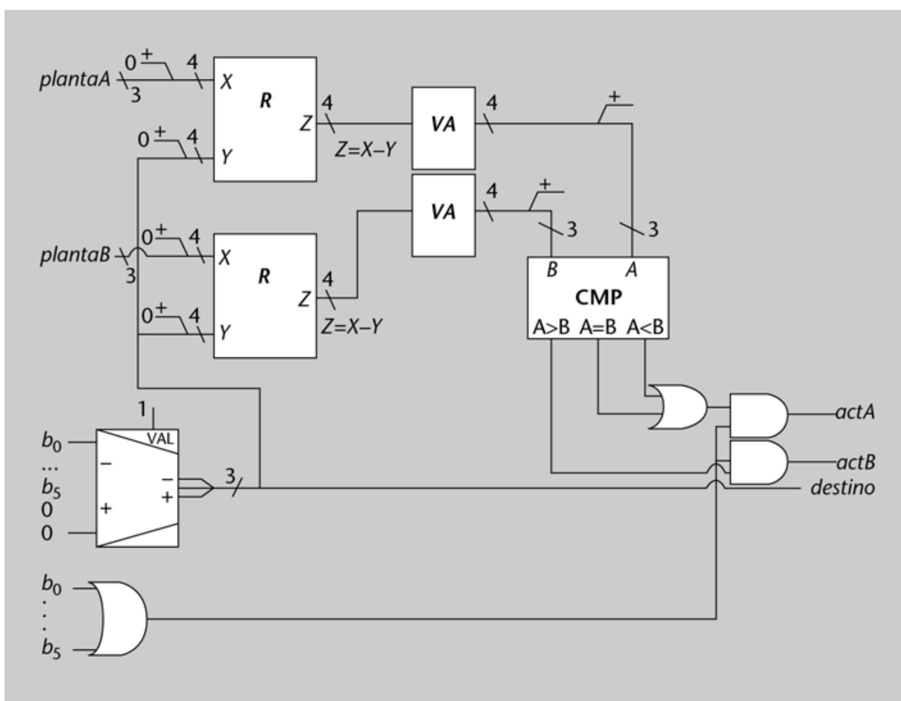
Las diferencias *plantaX – destino* pueden ser tanto positivas como negativas, porque el ascensor $X$ puede estar más arriba o más abajo que la planta desde donde se hace la llamada. La distancia que buscamos será el valor absoluto de la resta. Lo haremos utilizando un bloque $VA$, que contendrá el circuito que se ha diseñado en la actividad 57 (pero de 4 bits). Este valor absoluto nunca será mayor que 5, de manera que se puede representar con 3 bits. Por lo tanto, no se producirá nunca desbordamiento en este bloque $VA$, y es más, descartamos uno de sus bits de salida.

Una vez tenemos ya las dos distancias, podemos saber cuál de las dos es mayor mediante un comparador. Es suficiente con un comparador de números de 3 bits.

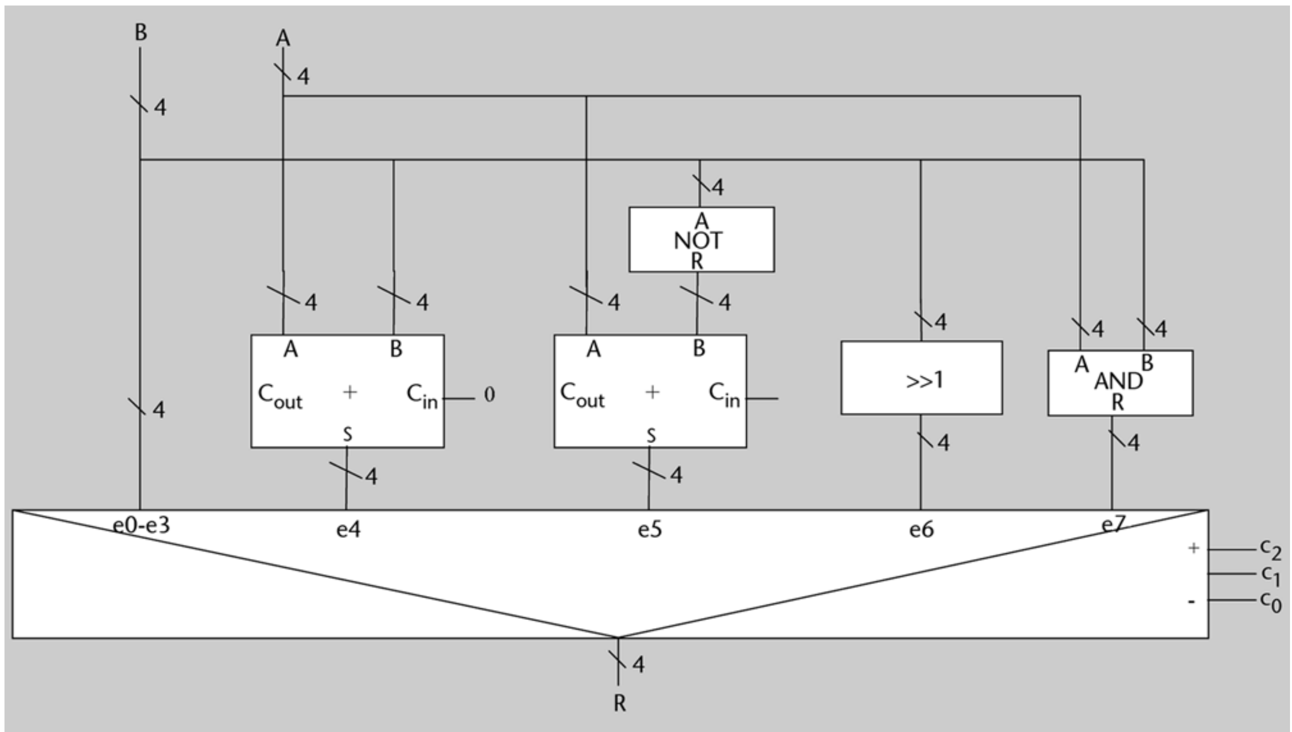
Si ponemos la distancia a la que se encuentra el ascensor $A$ en la entrada $A$ del comparador y la otra distancia en la entrada $B$, sabemos que cuando la salida $A < B$ del comparador sea 1 deberemos poner a 1 la señal *actA*. De manera análoga, cuando $A > B$ sea 1, tendremos que poner a 1 la señal *actB*. En caso de que $A = B$ sea 1, es decir, que los dos ascensores estén a la misma distancia, tenemos que poner a 1 *actA* (es decir, $actA = (A < B) + (A = B)$).

Puesto que las señales *actA* y *actB* se deben mantener a 0 mientras no se pulse ningún botón, hay que hacer un producto lógico (AND) entre estas salidas del comparador y una señal que valga 0 siempre que no se haya pulsado ningún botón. Lo podemos obtener, por ejemplo, haciendo una suma lógica (OR) con todos los bits $b_i$.

El circuito completo queda de la manera siguiente:



**13.** El circuito de esta UAL se muestra en la figura siguiente. Observamos que la entrada B se conecta a las cuatro entradas de menos peso del multiplexor.




## Bibliografía

**Gajsky, D.D.** (1997). *Principios de Diseño Digital*. Prentice-Hall.

**Hermida, R.; Corral, A. Del; Pastor, E.; Sánchez, F.** (1998). *Fundamentos de Computadores*. Madrid: Síntesis.

# Los circuitos lógicos secuenciales

Montse Peiron Guàrdia
Fermín Sánchez Carracedo

PID_00163600

Universitat Oberta de Catalunya
www.uoc.edu

# Índice

**Introducción** .......... 5

**Objetivos** .......... 6

**1. Caracterización de los circuitos lógicos secuenciales** .......... 7
1.1. Necesidad de memoria en los circuitos lógicos .......... 7
1.2. Reloj. Sincronización .......... 7

**2. El biestable D** .......... 10
2.1. Dispositivo elemental de memoria. El biestable D .......... 10
2.2. Señal de carga .......... 13
2.3. Entradas asíncronas .......... 14

**3. Bloques secuenciales** .......... 18
3.1. Registro .......... 18
3.2. Banco de registros .......... 23
3.3. Memoria RAM .......... 25

**4. El modelo de Moore** .......... 29
4.1. Estado. Transiciones .......... 29
4.2. Representación gráfica: grafos de estado .......... 34
4.2.1. Mecánica de diseño .......... 35
4.2.2. Notación .......... 37
4.2.3. Circuitos sin entradas .......... 38
4.3. Sincronización .......... 41
4.4. Implementación .......... 45

**Resumen** .......... 50

**Ejercicios de autoevaluación** .......... 51

**Solucionario** .......... 56

**Bibliografía** .......... 75

# Introducción

En el módulo “Los circuitos lógicos combinacionales” se ha visto que los circuitos computan funciones lógicas de las señales de entrada: el valor de las señales de salida en un instante determinado depende del valor de las señales de entrada en ese mismo momento. Cuando las señales de entrada varían, entonces, como consecuencia, también variarán las de salida (después del retraso producido por las puertas y bloques, que en este curso no tenemos en cuenta).

Ahora bien, en algunas aplicaciones es preciso que el valor de las señales de salida no dependa sólo del valor de las entradas en el mismo momento, sino que también tenga en cuenta los valores que las entradas han tomado anteriormente. En los circuitos que hemos conocido hasta ahora, esto no es posible: son necesarios elementos que tengan alguna “capacidad de recordar”, que son los que conformen los circuitos lógicos secuenciales.

En este módulo conoceremos el concepto de *sincronización* y se estudiarán los *biestables*, que son los dispositivos secuenciales más básicos, y los bloques secuenciales, que se construyen a partir de los biestables y tienen una funcionalidad determinada.

Después se presentará una de las maneras de formalizar el funcionamiento de un circuito temporal, el llamado *modelo de Moore*. El modelo de Moore utiliza los conceptos de *estado* y de *transiciones entre estados* para describir la evolución temporal del funcionamiento de un circuito temporal, que se representa gráficamente mediante un *grafo de estados*.

Este módulo permite describir el comportamiento de muchos circuitos secuenciales.

# Objetivos

El objetivo fundamental de este módulo es conocer los circuitos lógicos secuenciales, es decir, saber cómo están formados y utilizarlos con agilidad. Para llegar a este punto se tendrán que haber satisfecho los siguientes objetivos:

1. Saber discernir, a partir de la funcionalidad que se quiere que tenga un circuito lógico, si el circuito tiene que ser de tipo secuencial o combinacional.

2. Entender el concepto de memoria, la necesidad de una sincronización en los circuitos lógicos secuenciales y el funcionamiento de la señal de reloj.

3. Conocer el funcionamiento del biestable D y de todas las entradas de control que puede tener.

4. Conocer la funcionalidad de los diferentes bloques secuenciales, y saberlos utilizar en el diseño de circuitos.

5. Comprender los conceptos de *estado* y de *transición entre estados*. Entender todos los elementos de un grafo de estados, y ser capaces de construir el grafo de estados de un circuito cualquiera a partir de la descripción de su funcionalidad.

6. Saber deducir la evolución temporal de un circuito a partir del grafo de estados que describe el funcionamiento.

# 1. Caracterización de los circuitos lógicos secuenciales

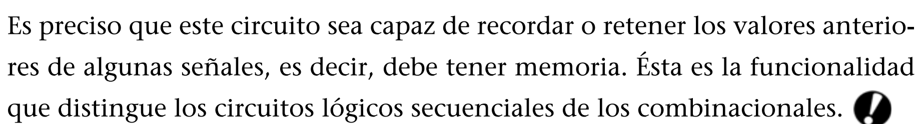
## 1.1. Necesidad de memoria en los circuitos lógicos

Sea un circuito con una señal de entrada $X$ y una de salida $Z$, ambos de $n$ bits, que interpretamos como números representados en complemento a dos. Supongamos que queremos que $Z = X + 2$. Con los elementos estudiados en el módulo “Los circuitos lógicos combinacionales” sabemos cómo se tiene que hacer, incluso de muchas formas diferentes. Cuando el valor en la entrada $X$ varíe, entonces $Z$ también cambiará consecuentemente de valor.

Supongamos que queremos que el valor $Z$ corresponda a la suma de todos los valores que han estado presentes en la entrada $X$ durante un intervalo de tiempo determinado (durante el cual el valor de $X$ ha variado). Con los dispositivos lógicos que conocemos hasta ahora no podemos conseguirlo, porque cuando cambiamos el valor de $X$, el valor anterior ha “desaparecido”, por lo que ya no podemos utilizarlo para calcular la suma.

**Secuencial**

La denominación“secuencial” deriva justamente de la capacidad de recordar la secuencia de valores que toman las señales.



Es preciso que este circuito sea capaz de recordar o retener los valores anteriores de algunas señales, es decir, debe tener memoria. Ésta es la funcionalidad que distingue los circuitos lógicos secuenciales de los combinacionales.

## 1.2. Reloj. Sincronización

En los circuitos combinacionales, la única noción temporal que interviene es el presente. En cambio, en los circuitos secuenciales se tiene en cuenta la evolución temporal de las señales (y aparece, como se verá más adelante, la noción de futuro).

Ahora bien, en la descripción del circuito del ejemplo anterior, ¿qué quiere decir con exactitud que “todos los valores que han estado presentes en la entrada $X$ durante un intervalo de tiempo determinado”? La señal $X$ puede ir cambiando de valor de forma aleatoria en el tiempo: puede valer 13 durante cuatro nanosegundos, después 25 durante diez nanosegundos, después 0 durante un nanosegundo, etc. ¿Cómo puede determinar el circuito en qué momento “$X$ ha dejado de tener el valor antiguo” y “empieza a tener el valor nuevo”? Para poder determinarlo, el circuito debe tener un **mecanismo de sincronización**. En los circuitos secuenciales que estudiaremos en este módulo se utiliza una señal de reloj como forma de sincronización.

**Discretización**

El reloj discretiza el tiempo: en lugar de verlo como una dimensión continua, los circuitos lo ven como una secuencia de instantes.

El **reloj** es una señal que sirve para determinar los instantes en que un circuito secuencial “ve” el valor de las señales, o “es sensible”, y responde en consecuencia.

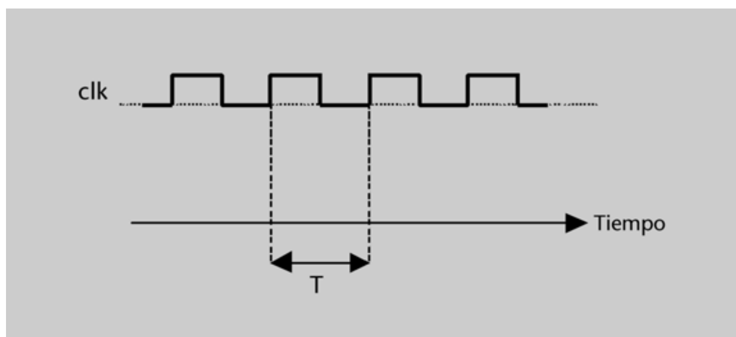
Esta tarea que lleva a cabo la señal de reloj se llama **sincronización de los circuitos**.

Concretamente, la señal del reloj toma los valores 0 y 1 de forma cíclica y continua desde la puesta en marcha de un circuito hasta que éste se detiene. De forma habitual, se utiliza la noción *clk* para hacer referencia a la señal del reloj (deriva del inglés *clock*).

La figura 1 muestra el cronograma de la señal de reloj. El ciclo que forma la secuencia de valores 0 y 1 tiene una duración determinada y constante, *T*, que se llama **periodo**. Se puede medir en segundos o, más habitualmente, en nanosegundos (mil millonésimas de segundo).

Tal como se ha visto en el módulo "Los circuitos lógicos combinacionales", las transiciones entre los valores 0 y 1 de una señal tienen un cierto retraso. Sin embargo, nosotros consideramos que los cambios de valor, de la señal de reloj o de cualquier otro son instantáneos.

Figura 1




**Generación de una señal de reloj**

Físicamente, la señal de reloj se genera a partir de cristales de cuarzo, un mineral que tiene la propiedad llamada *piezoelectricidad*: cuando recibe corriente eléctrica vibra con una frecuencia extremadamente grande y regular.

Los instantes en que la señal de reloj pasa de 0 a 1 se llaman **flancos ascendentes**. El intervalo que hay entre un flanco ascendente y el siguiente se llama **ciclo** o **ciclo de reloj**. Por tanto, la duración de un ciclo es un periodo de *T* segundos.

La **frecuencia** del reloj es la inversa del periodo, es decir, es el número de ciclos de reloj que ocurren durante un segundo. Se mide en hercios (ciclos por segundo); lo más habitual es usar el múltiplo gigahercios (mil millones de ciclos por segundo), que se abrevia GHz. Por ejemplo, si tenemos un reloj con un periodo de 0,75 nanosegundos, su frecuencia es la siguiente:

$$\frac{1 \text{ ciclo}}{0,75 \cdot 10^{-9} \text{ segundos}} = 1{,}33 \cdot 10^{9} \text{ ciclos/segundo} = 210^{-9} \text{ segundos} = 1{,}33 \text{ GHz}.$$

**Valores típicos de las frecuencias**

A principios del año 2010, los procesadores comerciales más rápidos tienen frecuencias de reloj de 3,5 GHz, con periodos cercanos a 0,3 nanosegundos. Sin embargo, la frecuencia base se puede multiplicar utilizando la técnica denominada *overclocking*.

La señal de reloj puede sincronizar los circuitos de varias formas. En este curso sólo se verá la que se usa de forma más habitual, llamada **sincronización por flanco ascendente**. Esta forma de sincronización establece que los dispositivos secuenciales de un circuito serán sensibles a los valores de la señal en los instantes de los flancos ascendentes, tal como veremos en el siguiente apartado. En el resto de los apuntes, si escribimos sólo *flanco* nos referimos a *flanco ascendente*.

**Otras formas de sincronización**

Otras formas de sincronización son por nivel 0, por nivel 1 y por flanco descendente.

## Actividades

**1.** Se quiere diseñar un sistema que reconozca si se produce la combinación 1010 en una entrada de cuatro bits. Indicad si el sistema es secuencial o combinacional y por qué.

**2.** Se quiere diseñar un sistema que reconozca una secuencia de cuatro dígitos decimales para identificar el número secreto de una tarjeta de crédito. El sistema tiene una entrada de datos única de cuatro bits, que codifican cada dígito. Indicad si el sistema es secuencial o combinacional y por qué.

**3.** ¿Cuál es el periodo del reloj de un procesador Intel Celeron E1200 que funciona a 1,6 GHz?

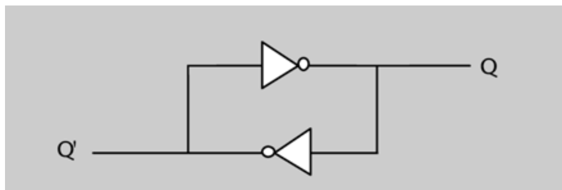
# 2. El biestable D

## 2.1. Dispositivo elemental de memoria. El biestable D

En el apartado anterior hemos visto la necesidad de que los circuitos lógicos tengan capacidad de memoria. En este apartado veremos cómo se construyen los dispositivos que pueden "recordar" los valores señalados.

Si examinamos el circuito que muestra la figura 2, vemos que el valor que hay en los puntos *Q* y *Q'* (0 ó 1) se mantendrá indefinidamente, ya que la salida de cada inversor está conectada con la entrada del otro. Por tanto, podemos decir que este circuito es capaz de "recordar", o mantener en el tiempo, un valor lógico.

Figura 2




Ahora bien, este circuito no es muy útil, porque no admite la posibilidad de modificar el valor recordado. Interesa diseñar un circuito que tenga esta misma capacidad de memoria, pero que además permita que el valor en el punto *Q* pueda cambiar según los requerimientos del usuario. Un circuito con estas características se llama *biestable*.

Los **biestables** son los dispositivos de memoria más elementales: permiten guardar un bit de información.

Un biestable tiene dos salidas, *Q* y *Q'*. Se dice que *Q* es "el valor que guarda el biestable" en cada momento, o "el valor almacenado en el biestable", y *Q'* es su negación.

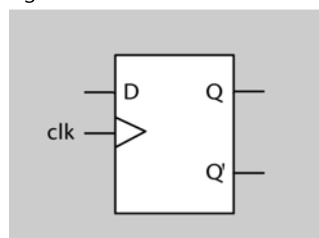
**La denominación *biestable***

Proviene del hecho de que el biestable puede estar "en dos estados": $Q = 0$ o $Q = 1$.

Hay diferentes tipos de biestables. En esta asignatura sólo veremos uno, el **biestable D**. La figura 3 muestra su representación gráfica. Podemos observar que tiene una entrada de reloj, ya que se trata de un dispositivo secuencial.

**La entrada de reloj de un dispositivo secuencial sincronizado por flanco ascendente se identifica con el símbolo >.**

Figura 3




**Implementación interna**

La implementación interna de un biestable D se basa en el circuito de la figura 2, pero no es objetivo de esta asignatura conocerla.

El biestable D funciona de la siguiente forma:

> La salida *Q* toma el valor que haya en la entrada *D* en cada flanco ascendente de reloj. Durante el resto del ciclo, el valor de *Q* no cambia.

Es decir, el biestable sólo es sensible al valor presente en la entrada *D* en los instantes de los flancos ascendentes.

La figura 4 muestra la tabla de verdad que describe el comportamiento del biestable D (no ponemos la columna correspondiente a *Q'* porque es la negación de *Q*). En esta figura se introducen algunas notaciones que se usarán de ahora en adelante:

> - El símbolo ↑ representa un flanco ascendente de reloj.
> - El símbolo $^{+}$ a la derecha del nombre de una señal se refiere al valor que tomará esta señal cuando se produzca el próximo flanco ascendente de reloj.

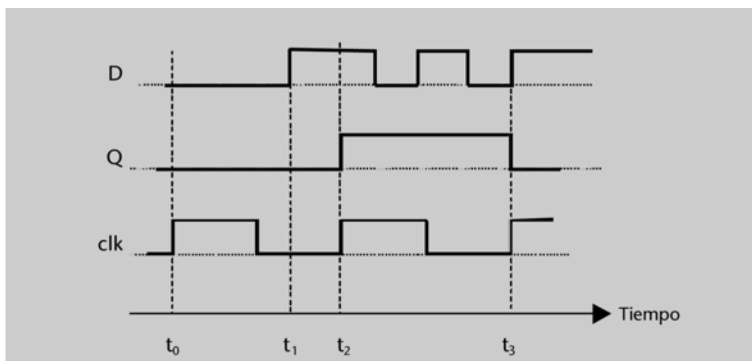
Por tanto, $Q^{+}$ no identifica ninguna señal del circuito, sino el valor que tendrá la señal *Q* en un instante futuro: a partir del momento en que se produce el próximo flanco. Esta notación, pues, nos permite describir con precisión la evolución temporal de las señales en un circuito lógico secuencial.

Figura 4

| D | clk | $Q^{+}$ |
|---|---|---|
| 0 | ↑ | 0 |
| 1 | ↑ | 1 |

La figura 5 muestra el cronograma del comportamiento de un biestable D durante dos ciclos, el que va del instante $t_0$ al instante $t_2$ y el que va de $t_2$ a $t_3$.

Figura 5




**Nota**

Notemos que no tendría sentido el hecho de dibujar una línea con el nombre $Q^{+}$ en un cronograma, ya que $Q^{+}$ no corresponde a ninguna señal.

En la figura 5 podemos observar que *D* toma diferentes valores durante un ciclo. En el ciclo que va de $t_0$ a $t_2$ vale primero 0 y después 1, y en un ciclo que va de $t_2$ a $t_3$ toma valores 1, 0, 1 y 0, y pasa de nuevo a 1 en el instante $t_3$ (coincidiendo con el tercer flanco ascendente del reloj). Nos podríamos preguntar cuál de estos valores es el que se carga en el biestable cuando llega el flanco ascendente, en los instantes $t_2$ y $t_3$.

**Valor que se carga en el biestable**

En los circuitos reales, el valor que se carga en el biestable es el que está presente de forma estable en la entrada *D* un cierto intervalo de tiempo anterior al flanco, que depende del retraso en la subida y bajada de tensión de las señales y de la tecnología utilizada para construir el biestable.

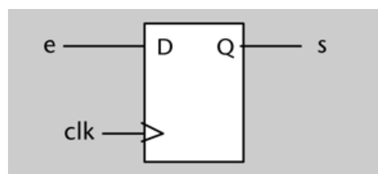
Para responder a esta pregunta, usaremos la siguiente convención: el valor que se carga en un biestable en el instante de un flanco es el que tiene la entrada *D* en el momento en que llega el flanco.

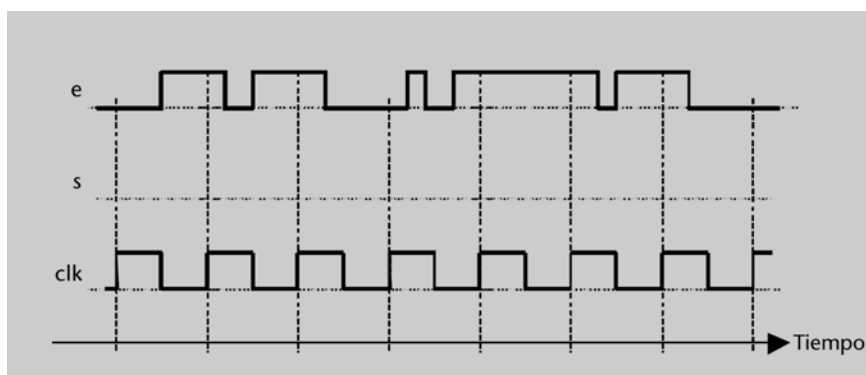
En un cronograma, esta idea se traduce en lo siguiente: el valor que se carga en el biestable en un flanco determinado es el que tiene la línea correspondiente a la entrada *D* cuando toca la línea vertical correspondiente a este flanco por la izquierda (ya que, en un cronograma, el tiempo transcurre de izquierda a derecha).

En la figura 5 podemos ver que en el instante $t_2$, *Q* toma el valor 1 (en este caso no hay duda), y en el instante $t_3$, *Q* toma el valor 0. La convención anterior nos dice que el valor que toma *Q* es 0, porque es el que hay en la línea correspondiente en *D* cuando ésta toca el flanco por la izquierda.

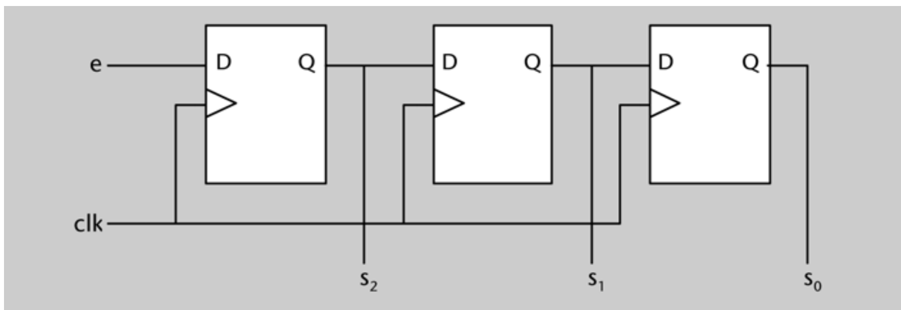
### Actividades

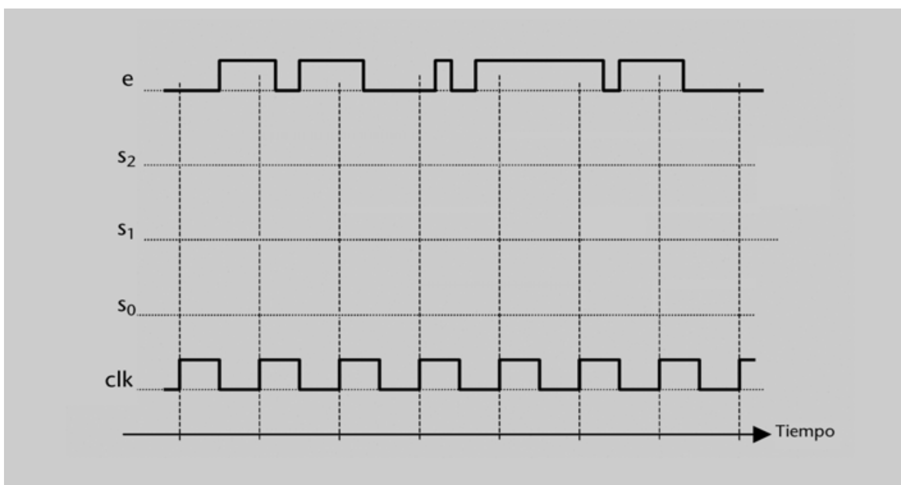
**4.** Completad el cronograma que corresponde al circuito de la figura.




**5.** Completad el cronograma que corresponde al circuito de la figura.



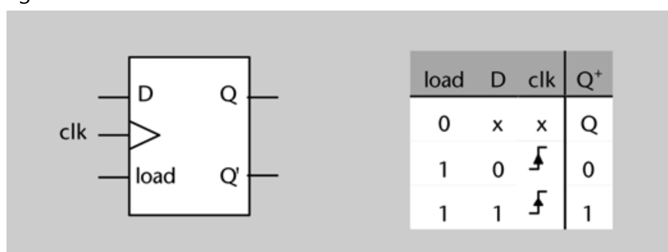

## 2.2. Señal de carga

Cuando se pone un biestable dentro de un circuito con otros componentes, a veces nos interesará que el contenido del biestable no cambie, ni siquiera en los instantes de los flancos, aunque varíe el valor de *D*; queremos que el biestable sea "insensible" a las variaciones de *D* cuando así lo requiramos.

Con este fin se añade al biestable una **señal de carga** que funciona de la siguiente forma: si vale 0, el valor del biestable no cambia. Si vale 1, el biestable funciona tal como se ha explicado en el apartado anterior.

La figura 6 muestra la representación gráfica de un biestable *D* con señal de carga (se identifica por la palabra *load*), y la tabla de verdad de su funcionamiento. Como es habitual, las *x* indican valores cualesquiera de las señales.

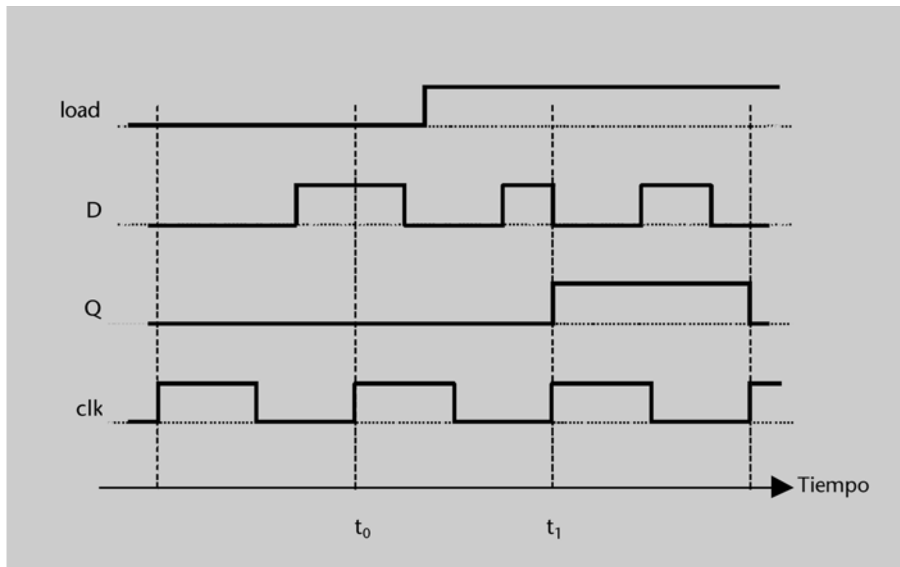
Figura 6




| load | D | clk | $Q^+$ |
|---|---|---|---|
| 0 | x | x | Q |
| 1 | 0 | ↑ | 0 |
| 1 | 1 | ↑ | 1 |

La figura 7 muestra el cronograma del comportamiento de un biestable D con señal de carga. Podemos ver que en el instante $t_0$ el valor de $Q$ no varía, aunque $D = 1$, porque la señal *load* está en 0. Cuando *load* = 1, entonces el biestable funciona tal como se ha estudiado en el apartado anterior.

Figura 7




### 2.3. Entradas asíncronas

El valor de un biestable *D* puede variar en los instantes de flancos ascendentes según el valor que hay en las entradas *D* y *load*. Ahora bien, se debe tener la capacidad de darle un valor inicial: el valor que tomará cuando se ponga en marcha un circuito.

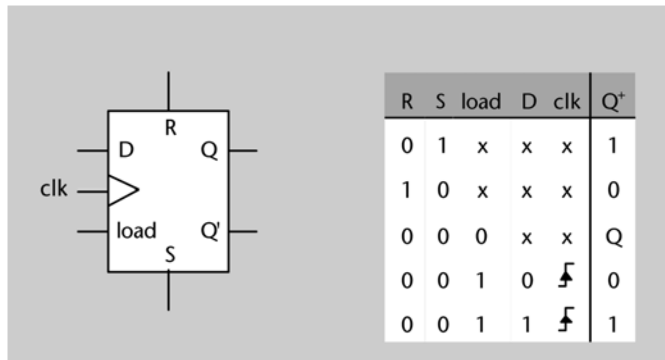
> Las **entradas asíncronas** de un biestable permiten modificar su valor de forma instantánea, de forma independiente al valor de la señal de reloj y de las entradas *D* y *load*. Se dice que las entradas asíncronas tienen más prioridad que el resto de las entradas.

Los biestables suelen tener dos entradas asíncronas:

- *R* (del inglés *reset*): en el momento en que se pone a 1, el biestable se pone a 0.
- *S* (del inglés *set*): en el momento en que se pone a 1, el biestable se pone a 1.

La figura 8 muestra la representación gráfica de un biestable *D* con entradas asíncronas y señal de carga, y la tabla de verdad que describe su comportamiento.

Figura 8

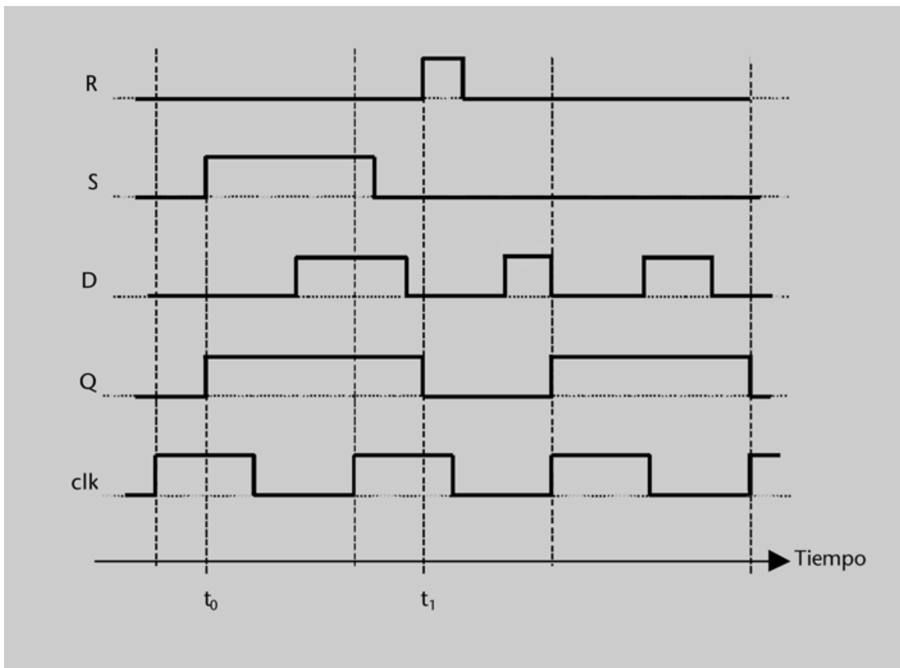



| R | S | load | D | clk | $Q^+$ |
|---|---|---|---|---|---|
| 0 | 1 | x | x | x | 1 |
| 1 | 0 | x | x | x | 0 |
| 0 | 0 | 0 | x | x | Q |
| 0 | 0 | 1 | 0 | ↑ | 0 |
| 0 | 0 | 1 | 1 | ↑ | 1 |

**Si tanto *R* como *S* valen 1, el comportamiento del biestable no se puede predecir. En los circuitos reales siempre se garantiza que esta situación no se produzca nunca.**

La figura 9 muestra el cronograma del comportamiento de un biestable D con entradas asíncronas y señal de carga (para simplificar el dibujo, supongamos que *load* = 1 todo el tiempo). Se puede observar que el biestable se pone a 1 en el instante $t_0$, aunque éste no coincida con un flanco ascendente, y se mantiene a 1 mientras *S* = 1, de forma independiente al valor de *D*. De igual forma, se pone a 0 en el instante $t_1$. En cambio, mientras ambas entradas asíncronas están a 0, el biestable modifica su valor sólo en los momentos de un flanco ascendente, de acuerdo con el valor de *D*.

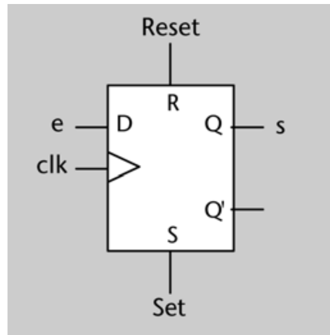
Figura 9

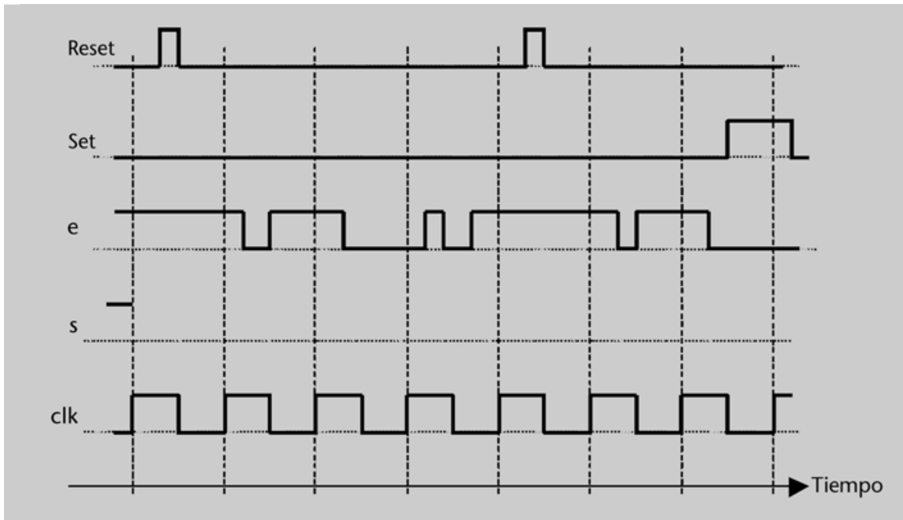



Cuando no dibujamos en un circuito las señales *load*, *R* y *S* de un biestable, asumiremos por defecto que valen 1, 0 y 0 respectivamente.

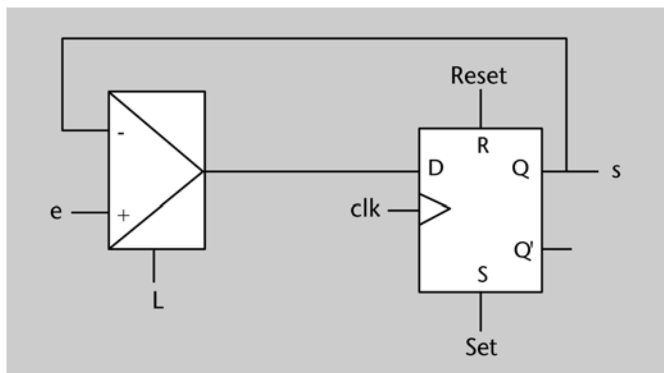
## Actividades

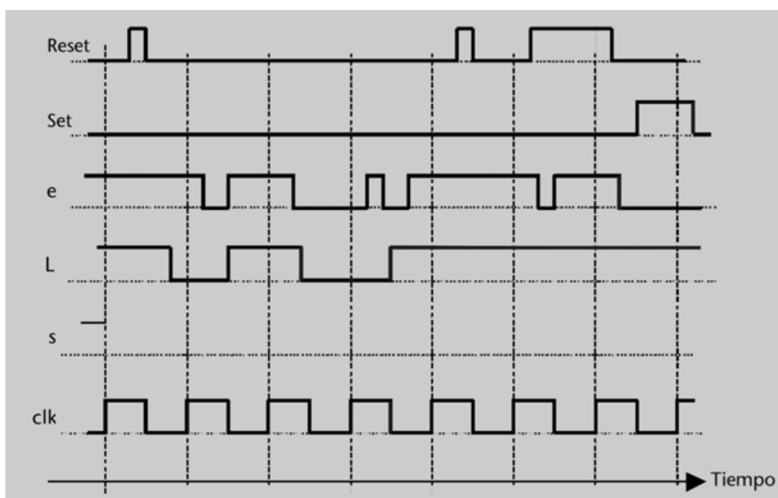
**6.** Completad el cronograma que corresponde al circuito de la figura suponiendo que en su inicio la salida Q vale 1.




7. Completad el cronograma que corresponde al circuito de la figura, suponiendo que inicialmente la salida *Q* vale 1. ¿Cuál es el papel de la señal *L* en el circuito?



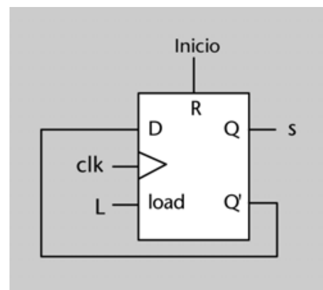



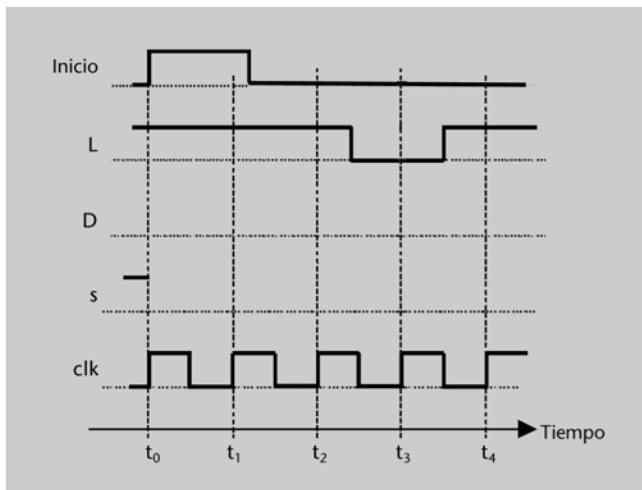
### Inicialización de los circuitos

Los circuitos secuenciales suelen tener una señal que actúa de forma asíncrona y tiene por misión inicializar el circuito. Esta señal, que llamaremos *Inicio*, está conectada a las entradas asíncronas de los biestables (en *R* o *S* según si el valor inicial tiene que ser 0 ó 1). Cuando el circuito se pone en funcionamiento, la señal *Inicio* vale 1 durante un cierto intervalo de tiempo, y después baja a 0 (se dice que **genera un pulso** a 1); un pulso siempre dura más de un ciclo de reloj (es decir, siempre se produce al menos un flanco ascendente mientras *Inicio* vale 1). Durante el funcionamiento normal del circuito, *Inicio* permanece a 0. Si en algún otro momento *Inicio* hace otro pulso a 1, el circuito se reinicializa.

## Actividades

**8.** Completad el cronograma que corresponde al circuito de la figura.



# 3. Bloques secuenciales

## 3.1. Registro

Hemos visto que un biestable permite guardar el valor de un bit. Para guardar el valor de una palabra de *n* bits serán necesarios *n* biestables D.

> Un **registro** es un bloque secuencial formado por *n* biestables D, que permite guardar el valor de una palabra de *n* bits.

La figura 10 muestra la representación gráfica de un registro. Se puede ver que presenta las siguientes señales:

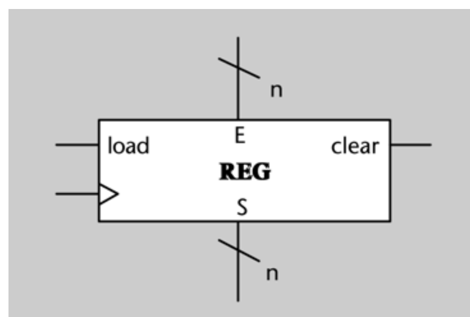
- Una entrada de datos de *n* bits, *E*. Cada uno de los bits de este bus está conectado con la entrada *D* de uno de los *n* biestables que forman el registro.

- Una salida de datos de *n* bits, *S*, que es un bus formado por las salidas *Q* de los *n* biestables que forman el registro.

- Dos entradas de control de un bit, *load* y *clear*. Estas dos señales están conectadas respectivamente a la señal *load* y a la entrada asíncrona *R* de cada uno de los biestables del registro.

- Una entrada de reloj, conectada a las entradas de reloj de todos los biestables.

Figura 10



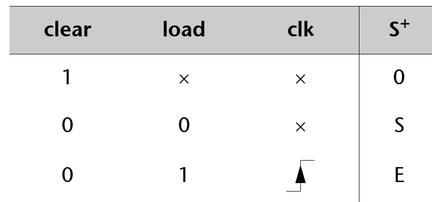

El funcionamiento del registro es el siguiente:

- La señal *clear* sirve para poner el contenido del registro a 0. Dado que se conecta con las entrada *R* de los biestables, es una señal asíncrona, es decir, actúa independientemente del reloj, y es el más prioritario, de forma que cuando está a 1, los *n* bits del registro se ponen a 0, independientemente del valor de las otras señales.

- Cuando *clear* está a 0, entonces los *n* biestables que forman el registro se comportan como *n* biestables D con señal de carga.

Este funcionamiento se puede expresar mediante la siguiente tabla de verdad:

**Si en un registro no dibujamos las señales *load* o *clear*, asumiremos que están a 1 y a 0, respectivamente.**



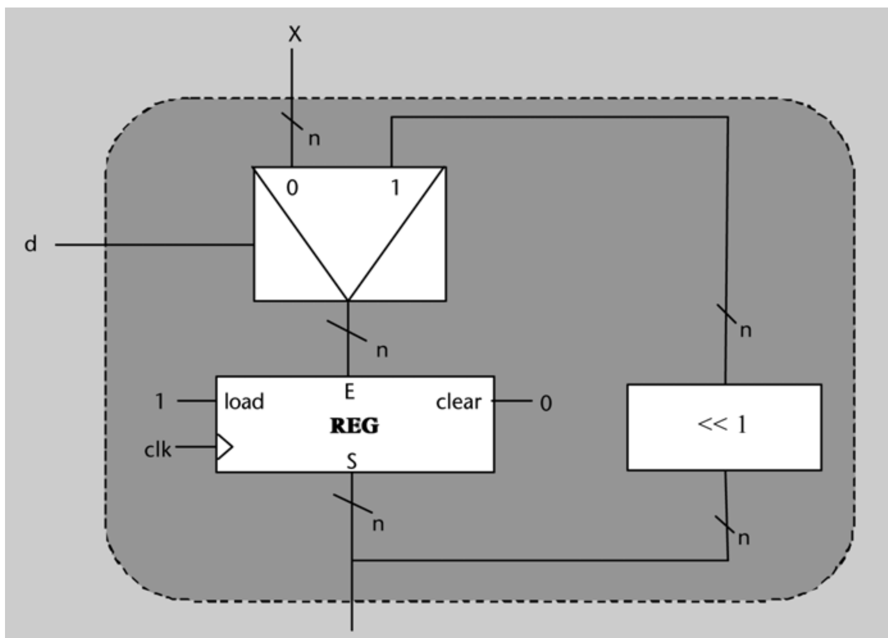
| clear | load | clk | $S^+$ |
|---|---|---|---|
| 1 | × | × | 0 |
| 0 | 0 | × | S |
| 0 | 1 | ↑ | E |

Cuando modificamos el valor de un registro haciendo que se cargue con el valor que hay en la entrada *E*, decimos que hacemos una **escritura** en el registro.

Cuando analizamos el contenido de un registro a partir de la salida S, decimos que hacemos una **lectura**.

A partir de un registro y bloques combinacionales, se pueden diseñar circuitos con una funcionalidad determinada. Por ejemplo, el circuito de la figura 11 permite que el registro se pueda cargar con el valor de la entrada *X* o que pueda desplazar su contenido 1 bit a la izquierda, en función de la señal *d*.

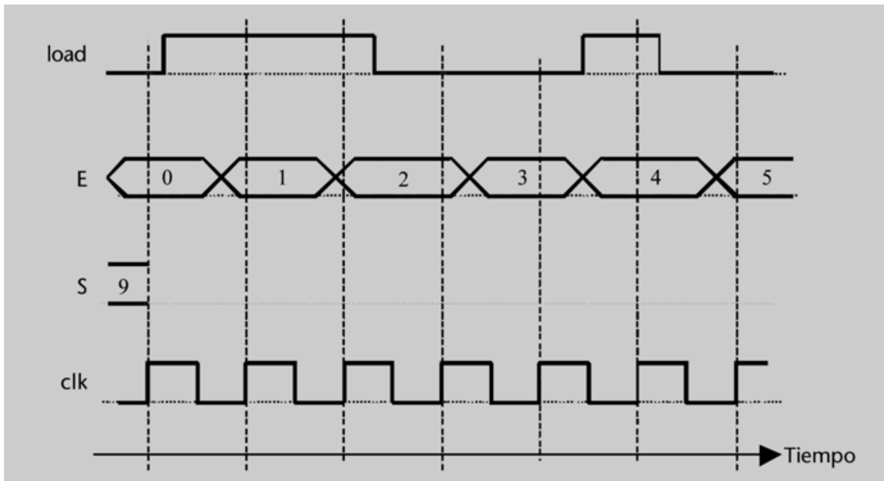
Figura 11




## Actividades

**9.** La siguiente figura muestra los valores de las señales *E* y *load* de un registro de ocho bits durante cierto intervalo de tiempo. Indicad en la línea etiquetada como *tiempo* los

instantes en que el registro se carga con la entrada *E* y la secuencia de valores que tomará la salida *S* del circuito. Observad que inicialmente el valor de *S* es 9.

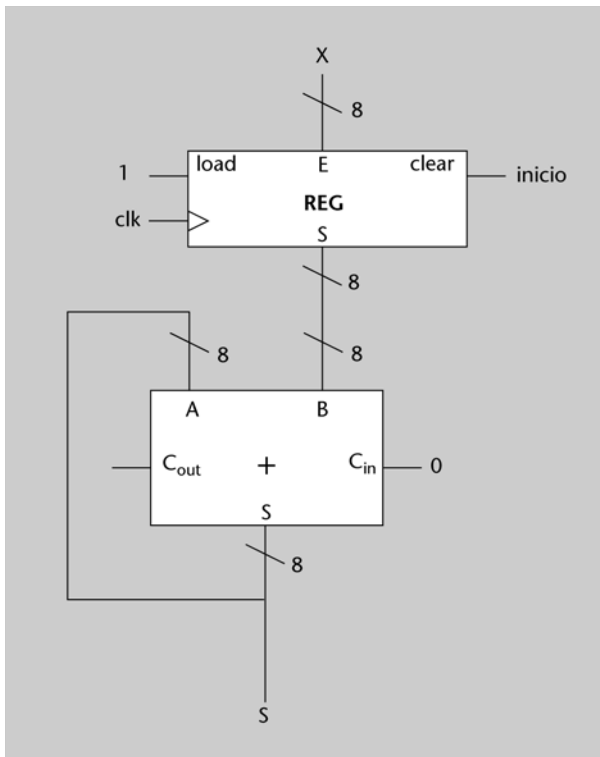



**Interpretación de un cronograma**

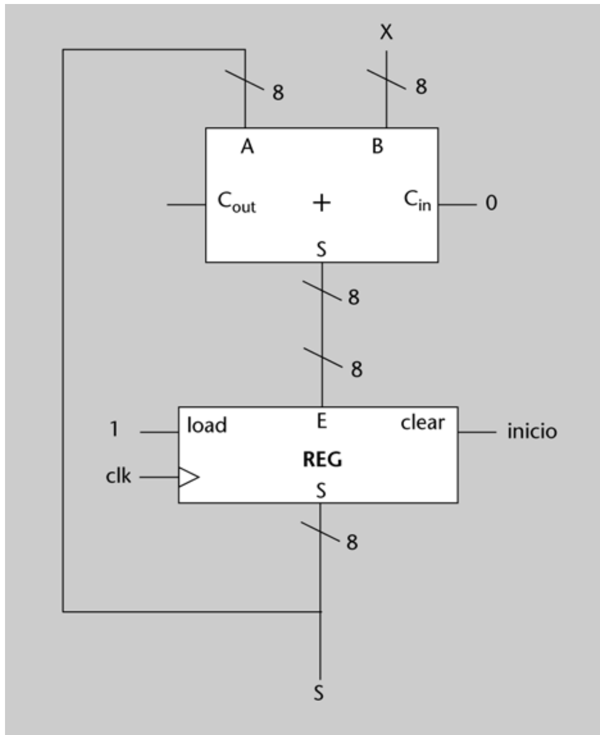
En un cronograma, los valores de palabras de *n* bits se dibujan mediante un hexágono, en cuyo interior se escribe el valor de la señal en decimal. Los puntos de contacto entre los extremos de los hexágonos indican el momento en que la palabra cambia de valor.

**10.** Se quiere diseñar un circuito con una entrada *X* de 8 bits que codifica un número natural en binario, y una salida *S* también de 8 bits que muestre en todo momento la suma acumulada de los valores que ha tenido *X* desde la inicialización del circuito y hasta el ciclo presente.

**a)**Un diseñador inexperto propone este circuito. ¿Por qué no es válido?



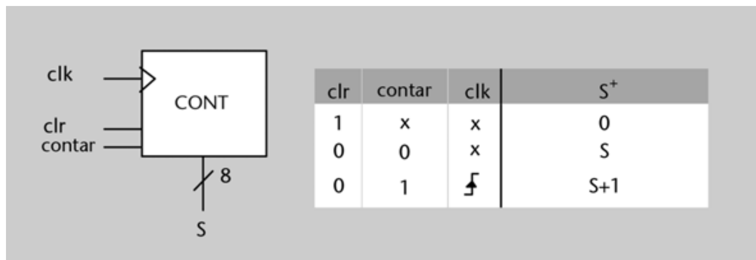

**b)**Un diseñador con un poco más de experiencia propone este otro circuito. ¿En qué mejora al anterior? Este circuito también tiene un inconveniente, ¿cuál es? ¿Es posible evitarlo?




**11.** Diseñad un bloque secuencial contador, CONT, con una salida *S* de 8 bits. Esta salida es un número natural (codificado en binario) que, siempre que la señal de entrada *contar* valga 1, se incrementará en cada flanco del reloj según esta expresión:

$$S^+ = (S + 1) \bmod 256$$

(por tanto, después de valer 255 pasa a valer 0). El bloque tiene, además, una entrada asíncrona, *clr*, que cuando vale 1 pone la salida a 0. El funcionamiento del circuito se muestra en la siguiente figura.




**12.** Diseñad un circuito secuencial con una entrada de datos *E* de 8 bits, dos entradas de control $c_1$ y $c_0$ de 1 bit y una salida *S* de 8 bits. *E* y *S* codifican números naturales en binario. El funcionamiento del circuito viene descrito por la tabla siguiente:

| $c_1$ | $c_0$ | *clk* | $S^+$ |
|---|---|---|---|
| 0 | 0 | ↑ | 0 |
| 0 | 1 | ↑ | *E* |
| 1 | 0 | ↑ | (*S*+1) mod 256 |
| 1 | 1 | ↑ | (*S*+*E*) mod 256 |

Inicialmente, *S* debe tener el valor 0. El circuito no puede contener más de un bloque sumador.

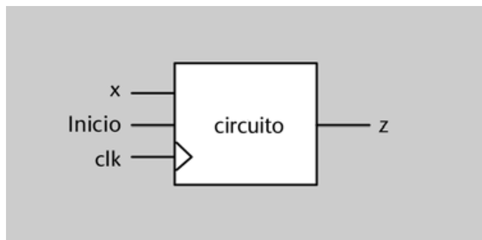
**13.** Se quiere diseñar un circuito que controle el estado de ocupación de un aparcamiento, en concreto poniendo en verde un semáforo mientras queden plazas libres y ponién-

dolo en rojo cuando esté lleno. El semáforo se pone verde cuando la señal *lleno* vale 0 y se pone rojo cuando vale 1.

El aparcamiento tiene capacidad para 500 vehículos. Cuando entra un coche, se produce un pulso en la señal *entra*, y cuando sale un coche se produce un pulso en la señal *sale*. Otros sistemas de control del aparcamiento garantizan que en un mismo ciclo nunca entrará y saldrá un coche al mismo tiempo, y generan un pulso en la señal *Inicio* cuando el aparcamiento abre las puertas, momento en el que no habrá ningún coche.

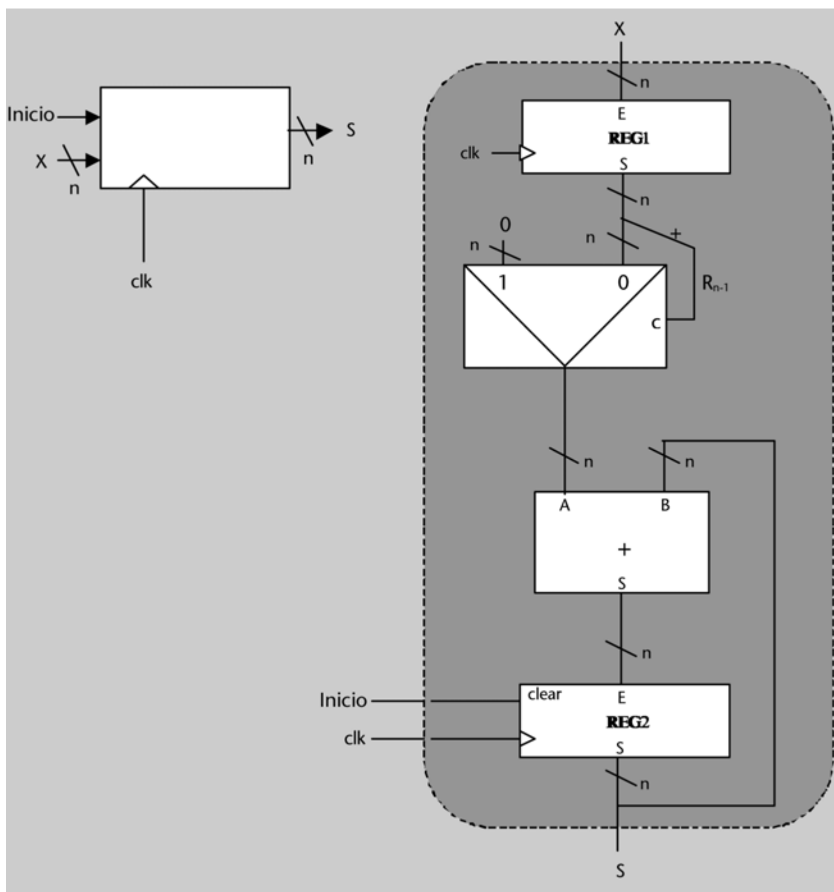
**a)**¿Qué señales de entrada y de salida debe tener el circuito? ¿De cuántos bits cada una?
**b)**El circuito debe ser secuencial, ¿por qué?
**c)**Diseñad el circuito e indicad el ancho de todos los buses.

**14.** Utilizando un registro de cuatro bits, bloques combinacionales y puertas, diseñad un circuito secuencial que reconozca si se ha producido la secuencia de valores 1010 en una entrada *x* de 1 bit. Los diferentes valores de *x* son los que tenga esta señal al llegar cada flanco de reloj. Cuando reconoce la secuencia, el circuito tiene que poner la señal de salida *z* a 1. El circuito tiene otra señal de entrada, *Inicio*, que genera un pulso a 1 para inicializar el circuito.




**15.** Analizad qué hace el circuito de la figura, teniendo en cuenta lo siguiente:

– *X* y *S* son números enteros representados en complemento a 2.
– La entrada *X* tiene un valor nuevo en cada ciclo de reloj.
– $R_{n\text{-}1}$ se refiere al bit de más peso del registro REG1.
– La señal *Inicio* funciona de la forma habitual.

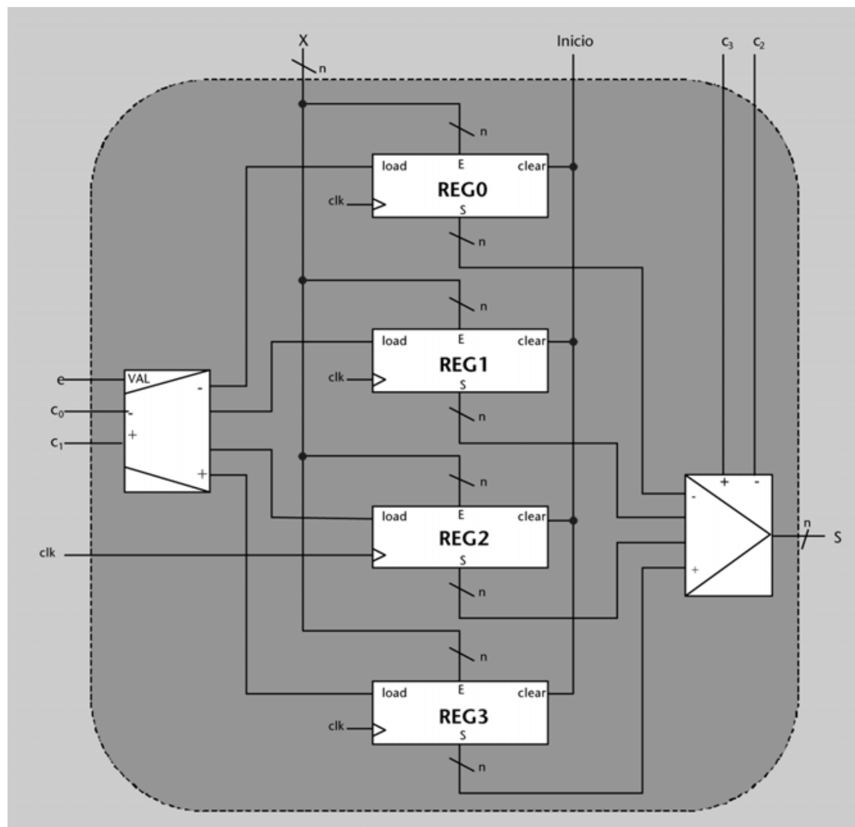



**Precisiones sobre los gráficos**

En los circuitos secuenciales, supondremos siempre que hay una señal única de reloj (*clk*). Sin embargo, en las figuras a veces no se conectan todas las entradas de reloj con una misma línea, con el fin de aclarar el dibujo.

En general, si en un circuito hay más de un punto identificado por un mismo nombre de señal, se entiende que los puntos están conectados, aunque no estén unidos por una línea. Por ejemplo, en la figura de la izquierda se escribe dos veces "clk", pero las dos corresponden a la misma señal.

**16.** Analizad qué hace el circuito de la figura siguiente:




**Intersección de cables**

A veces, en el diseño de circuitos se dibujan puntos en las intersecciones de cables para esclarecer el diseño.

**17.** Se quiere diseñar un circuito que controle la entrada a un edificio de acceso restringido, a cuya puerta hay un dispositivo con cinco teclas numéricas (del 0 al 4) y una tecla Entrar. Quien quiera entrar en el edificio debe teclear un código en el teclado numérico y pulsar Entrar; esto generará un pulso a 1 en la señal *entrar*, de un ciclo de reloj de duración, y pondrá en la señal *código* la representación en binario del código que se acaba de teclear. Se puede pulsar cualquier número de teclas numéricas, pero sólo se tendrán en cuenta las tres últimas que se hayan tecleado antes de pulsar *Entrar*.
Si el código coincide con la contraseña correcta, se enciende una luz verde (se consigue activando la señal *verde*) y la puerta se abre. Si el código es diferente de la contraseña, se enciende una luz amarilla (se consigue activando la señal *amarillo*) y la puerta no se abre. Si se teclea un código equivocado tres veces seguidas, se enciende una luz roja (se consigue activando la señal *rojo*) y la puerta se bloquea hasta que venga el portero y reinicialice el sistema (generando un pulso en la señal *Inicio*).
Las luces se deben apagar un cierto número de segundos después de que se hayan encendido, y también se tienen que mantener apagadas mientras nadie haya tecleado ningún código. Sin embargo, de esto se ocupa otro subsistema (es decir, sólo os tenéis que preocupar de que cuando alguien haya tecleado un código se encienda la luz apropiada).

**a)** ¿Qué entradas y salidas tiene el circuito? ¿De cuántos bits cada una?
**b)** Diseñad el circuito, suponiendo que la contraseña correcta no varía nunca y que disponéis de un registro que estará siempre cargado con su codificación en binario. Indicad el ancho de todos los buses.
c) ¿Qué elementos sería necesario añadirle para permitir que se pudiera cambiar la contraseña?

## 3.2. Banco de registros

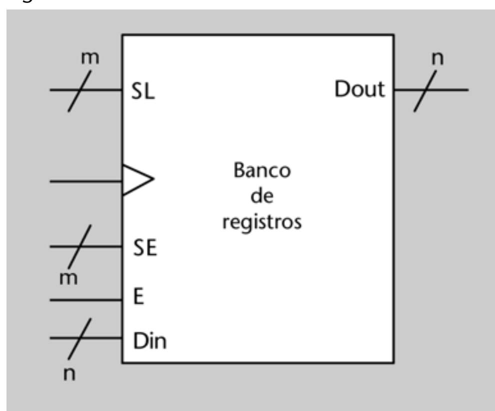
Un **banco de registros** es una agrupación de un cierto número de registros, todos del mismo número de bits. El contenido de los registros se puede leer y modificar gracias a los **puertos de lectura y de escritura**.

El número de registros de un banco siempre es una potencia de 2. Los $2^m$ registros están numerados desde 0 hasta $2^m - 1$.

La figura 12 muestra la representación gráfica de un banco de registros. Se puede ver que dispone de las siguientes señales:

- Una entrada de selección de lectura, *SL*, de *m* bits (si $2^m$ es el número de registros del banco.
- Una salida *Dout*, del mismo número de bits que los registros del banco. *SL* y *Dout* forman el **puerto de lectura del banco**.
- Una entrada de selección de escritura, *SE*, de *m* bits.
- Una entrada de permiso de escritura, *E*, de un bit.
- Una entrada *Din*, del mismo número de bits que los registros del banco. *SE, E* y *Din* forman el **puerto de escritura del banco**.

Figura 12




**Banco de registros**

En general, un banco de registros puede tener varios puertos de lectura y de escritura. El número de puertos de cada tipo determina el número de operaciones de lectura y de escritura que se pueden hacer de forma simultánea. Por ejemplo, si tiene dos puertos de lectura y uno de escritura, se pueden leer dos registros y escribir otro de forma simultánea. En esta asignatura, siempre tendrán un puerto de escritura y uno de lectura.

El banco de registros funciona de la siguiente forma:

- Para hacer una lectura, hay que poner en la entrada *SL* la codificación binaria del número de registro que se quiere leer. Entonces, el contenido de este registro estará presente en la salida *Dout*.
- Para hacer una escritura, hay que poner en *SE* la codificación binaria del número de registro que se quiere escribir y poner la entrada *E* a 1. Cuando se produzca el próximo flanco ascendente de reloj, el valor que haya en *Din* se escribirá en el registro indicado por *SE*.

**Atención**

Fijémonos en que en el banco "siempre se está leyendo"; dicho de otra forma, en *Dout* hay en todo momento el contenido del registro indicado por *SL*. En cambio, sólo se escribe en los ciclos en que $E = 1$; la escritura se realizará al final del ciclo, coincidiendo con el próximo flanco de reloj.

Como se puede observar, la entrada *E* funciona como una señal de carga: si está a 0 no se puede modificar el contenido de ningún registro del banco.

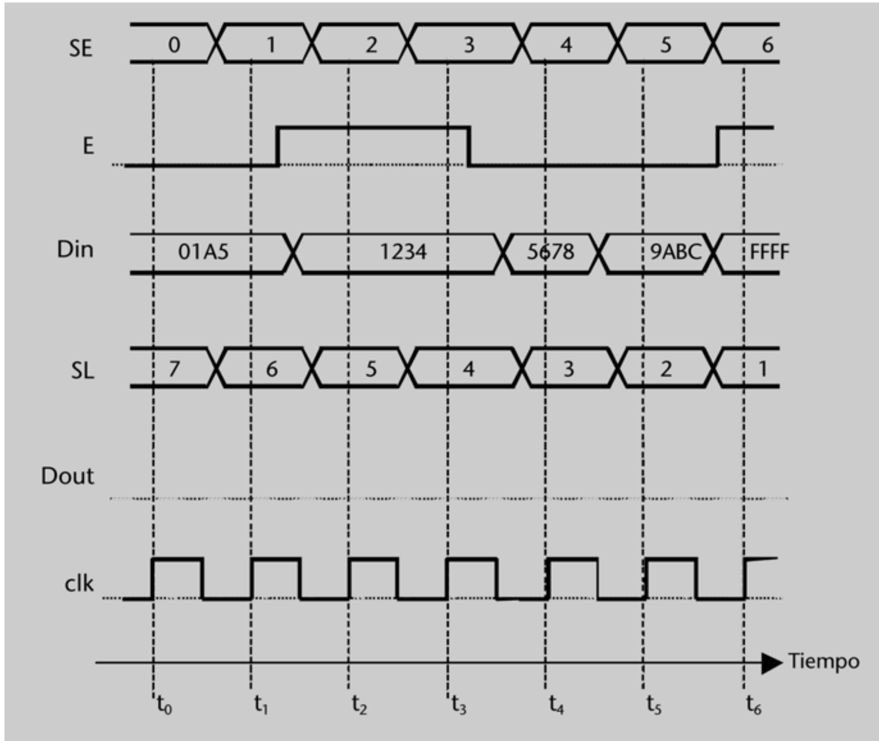
### Actividades

**18.** Se dispone de un banco de registros de 16 bits. A partir del cronograma que se muestra a continuación, haced lo siguiente:

**a)** Indicad qué registros se escriben, en qué momento y con qué valor.

**b)** Indicad en el cronograma el valor de la salida *Dout* suponiendo que el valor inicial de los registros, expresado en hexadecimal, sea el siguiente:

*R0* = 0000,
*R1* = 1111,
*R2* = 2222,
*R3* = 3333,
*R4* = 4444,
*R5* = 5555,
*R6* = 6666,
*R7* = 7777.




**c)** Indicad el valor de todos los registros del banco después del instante $t_6$.

**19.** Implementad un circuito con la misma funcionalidad que el de la actividad 16, pero sin la señal *Inicio*, utilizando sólo un banco de cuatro registros.

### 3.3. Memoria RAM

La **memoria RAM** es un bloque secuencial que permite guardar el valor de un cierto número de palabras ($2^m$) de un cierto número de bits ($n$).

**RAM**

La denominación RAM proviene del inglés *random access memory* ("memoria de acceso aleatorio"). Se le dio este nombre porque el tiempo que se tarda en hacer una lectura o una escritura no depende de la palabra a la que se acceda (a diferencia de lo que pasaba en otros dispositivos de memoria que se utilizan en los primeros computadores).

La funcionalidad de una memoria RAM, pues, es similar a la de un banco de registros. Las diferencias entre ambos bloques son las siguientes:

- El tamaño: un banco de registros suele guardar unas cuantas decenas de palabras, mientras que una memoria RAM puede guardar varios millones.
- La velocidad: por cómo se implementan físicamente una y otra, el tiempo de respuesta (retraso) de una memoria RAM es mucho mayor que el de un banco de registros (y, por tanto, este último es más rápido).

- La implementación interna de ambos bloques es muy diferente.

- En un banco de registros, las escrituras se hacen coincidiendo con los flancos ascendentes del reloj. En cambio, la memoria no tiene señal de reloj: las escrituras son efectivas un cierto intervalo de tiempo después de haber dado la orden de escribir.

En este curso no se estudiarán detalladamente estas cuestiones. Tenemos suficiente con la idea de que un banco de registros es pequeño y rápido, y una memoria RAM es grande y lenta. También asumiremos que la memoria RAM está sincronizada de la misma manera que los bancos de registros, con una señal de reloj.

Como en el caso de la memoria ROM que hemos visto en el módulo "Los circuitos lógicos combinacionales", la memoria RAM se puede ver como un archivador con cajones, numerados con una dirección. Cada cajón guarda una palabra. Llamamos *M*[*i*] a la palabra guardada en el cajón con dirección *i*.

La figura 13 muestra la representación de una memoria RAM con un solo **puerto de lectura/escritura**. Se puede ver que dispone de las siguientes señales:

**Nota**

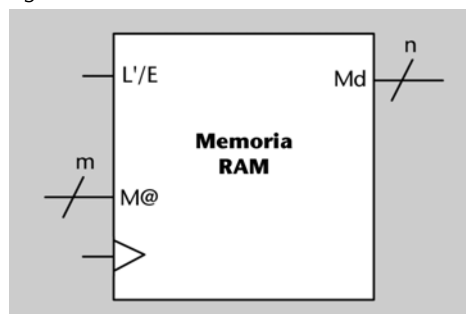
Las memorias también pueden tener un cierto número de puertos de lectura y de escritura, que determinan el número de operaciones que se pueden hacer de forma simultánea.
En una memoria con un solo puerto de lectura/escritura como la que presentamos en estos apuntes, en cada momento sólo se puede hacer o una lectura o una escritura.

- Una entrada de direcciones, *M@*. Si la memoria tiene capacidad para $2^m$ palabras, la entrada tendrá *m* bits.

- Una entrada/salida de datos, *Md*, de *n* bits (si las palabras que guarda la memoria son de *n* bits).

- Una entrada de control, *L'/E*, que indica en todo momento si se tiene que hacer una lectura o una escritura.

Figura 13




El funcionamiento de la memoria es el siguiente:

- Si *L'/E* = 0, entonces se hace una lectura: por el bus *Md* sale el valor de la palabra que está guardada en la dirección indicada por *M@*. Si *M@* cambia mientras *L'/E* = 0, *Md* variará también inmediatamente (en este curso, asumimos que las lecturas a memoria tienen retraso 0).

- Si *L'/E* = 1, entonces se hace una escritura: la palabra indicada por *M@* toma el valor que hay en *Md* en el primer flanco de reloj que se produzca después de activar *L'/E* (asumimos que las escrituras tampoco tienen retraso). Mientras *L'/E* = 1, el bus *Md* toma el valor 0 (a diferencia de lo que sucede en los bancos de registros, que "siempre están leyendo").

La siguiente tabla de verdad resume el funcionamiento de la memoria RAM.

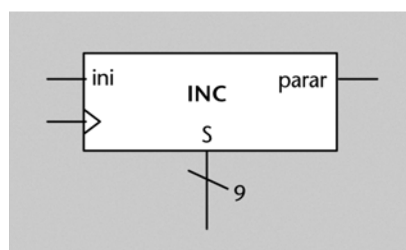
| L'/E | |
|---|---|
| 0 | Md := M[M@] |
| 1 | M[M@] := Md |

La capacidad de una memoria RAM se suele medir en *bytes* (palabras de ocho bits). Como hemos dicho, suele contener varios millones de palabras, y, por eso, para indicar su capacidad se suelen utilizar las letras k, M y G, que tienen los siguientes significados:

| Letra | Significado | Ejemplo |
|---|---|---|
| k | $2^{10} = 1.024 \cong 10^3$ | 16 kb (16 *kilobytes*) = $2^{14}$ *bytes* |
| M | $2^{20} = k \cdot k \cong 10^6$ | 32 Mb (32 *megabytes*) = $2^{25}$ *bytes* |
| G | $2^{30} = k \cdot M \cong 10^9$ | 2 Gb (2 *gigabytes*) = $2^{31}$ *bytes* |

## Actividades

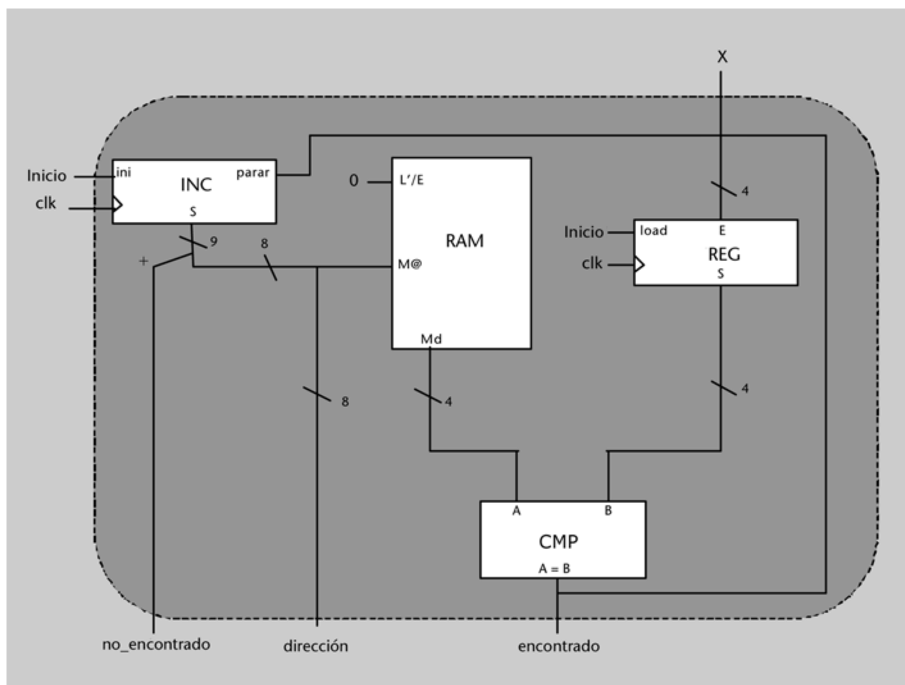
**20.**
**a)** Diseñad un bloque incrementador, INC, con una salida S de nueve bits (parecido al que se ha diseñado en la actividad 11), que funciona de la siguiente forma:




- Cuando la entrada *ini* está en 1, *S* se pone a 0.
- En cada flanco de reloj, *S* se incrementa en 1.
- Si en algún momento la señal *parar* vale 1, entonces *S* deja de incrementarse y mantiene su valor en la salida.
- Cuando *S* llega a 256 también deja de incrementarse, y su salida se mantiene en 256 hasta que en la entrada *ini* vuelva a haber un 1.

A partir del circuito de la siguiente figura (el bloque INC es el diseñado en el apartado *a*), contestad estas preguntas:

**b)** Indicad cuáles son las entradas y salidas del circuito, y cuántos bits tiene cada una.
**c)** ¿Qué bloques del circuito son combinacionales y cuáles son secuenciales?
**d)** ¿Cuál es el tamaño de la memoria RAM del circuito?
**e)** Indicad qué función hace este circuito y razonad la respuesta (recordad que *Inicio* genera un pulso a 1 cuando el circuito se pone en marcha).


X
Inicio
clk
ini
INC
parar
S
9
8
+
0
L'/E
RAM
M@
Md
4
Inicio
load
E
REG
clk
S
4
8
4
4
A
B
CMP
A = B
no_encontrado
dirección
encontrado

# 4. El modelo de Moore

## 4.1. Estado. Transiciones

El **modelo de Moore** es una forma de expresar o modelizar el funcionamiento de un circuito lógico secuencial. Es fundamental en los conceptos de *estado* y *transiciones entre estados*.

**Otras modelizaciones**

Otra forma muy usual de modelizar un circuito secuencial es el llamado **modelo de Mealy** (que no se estudia en esta asignatura).

Para introducir estos conceptos utilizaremos un ejemplo:

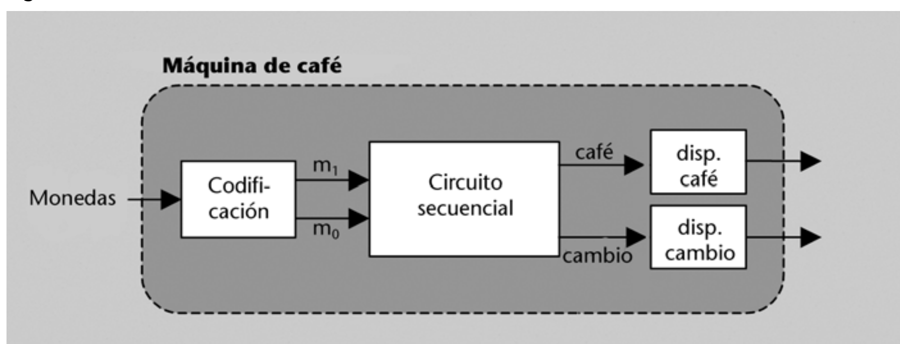
Imaginemos un circuito que controla el funcionamiento de una máquina expendedora de café. Para simplificar, asumiremos que la máquina sólo sirve un único tipo de café, y que sólo admite monedas de 0,5 euros y de 1 euro. La información sobre las monedas introducidas se codifica mediante dos señales lógicas $m_1$ y $m_0$, tal como se muestra en la tabla al margen. Las señales $m_1$ y $m_0$ serán las entradas del circuito.

Tabla 1

| Monedas introducidas | $m_1$ | $m_0$ |
|---|---|---|
| Ninguna moneda | 0 | 0 |
| Moneda de 0,5 euros | 0 | 1 |
| Moneda de 1 euro | 1 | 0 |

El precio del café es de 1,5 euros. La máquina tiene dos dispositivos de salida, uno para servir el café y otro para dar el cambio. Estos dos dispositivos están controlados respectivamente por las señales lógicas *café* y *cambio*, de forma que la máquina dará café o cambio cuando la señal correspondiente esté a 1. La figura 14 muestra el esquema del funcionamiento de la máquina.

Figura 14




Para determinar en cada momento si tiene que servir café o dar cambio, la máquina necesita saber cuánto dinero se le ha introducido hasta el momento. Se pueden dar las siguientes situaciones:

- Se han introducido 0 euros.
- Se han introducido 0,5 euros.
- Se ha introducido 1 euro.
- Se han introducido 1,5 euros.
- Se han introducido 2 euros.

Se llama **estado** a cada situación diferente en la que se puede encontrar un circuito.

En nuestro ejemplo, la máquina de café puede encontrarse en cinco estados diferentes, los descritos por las cinco posibilidades anteriores.

Las señales de salida del circuito tomarán un valor u otro según en qué estado se encuentre el circuito. En el caso de la máquina de café, la señal *café* estará a 1 cuando se hayan introducido al menos 1,5 euros, y la señal *cambio* estará a 1 si se han introducido 2.

Una **tabla de salida** se expresa mediante el valor que toman las señales de salida en cada estado.

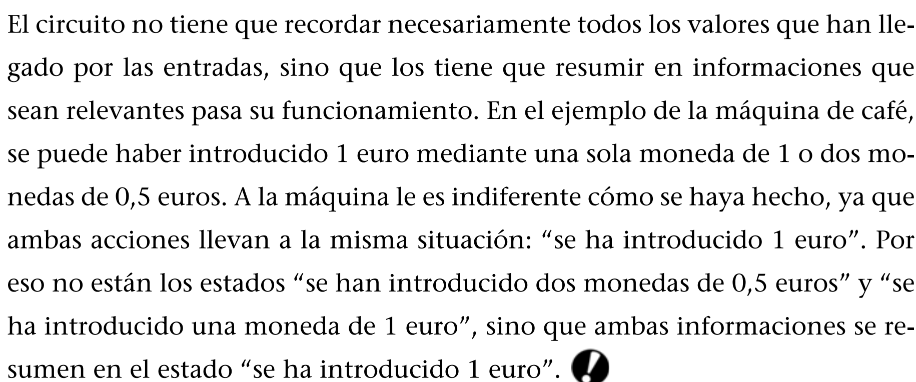
La tabla de salidas tiene a la izquierda los diferentes estados y a la derecha, el valor que toman las diferentes señales de salida en cada estado. La tabla de salidas del circuito de la máquina de café es la siguiente:

| Estado | café | cambio |
|---|---|---|
| Se han introducido 0 euros. | 0 | 0 |
| Se han introducido 0,5 euros. | 0 | 0 |
| Se ha introducido 1 euro. | 0 | 0 |
| Se han introducido 1,5 euros. | 1 | 0 |
| Se han introducido 2 euros. | 1 | 1 |

**Explicación de la tabla de salidas**

Cuando la señal *café* esté a 1, la máquina servirá un café. Cuando la señal *cambio* esté a 1 (es decir, cuando el estado sea "se han introducido 2 euros"), la máquina dará 0,5 euros de cambio (recordemos que un café vale 1,5 euros).

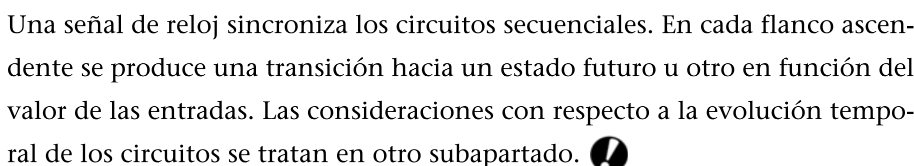
A medida que el tiempo avanza, el circuito cambia de estado en función de los valores que vayan llegando por las entradas. Mientras no se haya introducido ninguna moneda, la máquina de café se encontrará en el estado "se han introducido 0 euros". Cuando se introduzca una moneda de 0,5 euros, pasará al estado "se han introducido 0,5 euros"; si la moneda es de 1 euro pasará al estado "se ha introducido 1 euro".



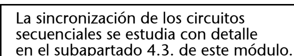
El circuito no tiene que recordar necesariamente todos los valores que han llegado por las entradas, sino que los tiene que resumir en informaciones que sean relevantes pasa su funcionamiento. En el ejemplo de la máquina de café, se puede haber introducido 1 euro mediante una sola moneda de 1 o dos monedas de 0,5 euros. A la máquina le es indiferente cómo se haya hecho, ya que ambas acciones llevan a la misma situación: "se ha introducido 1 euro". Por eso no están los estados "se han introducido dos monedas de 0,5 euros" y "se ha introducido una moneda de 1 euro", sino que ambas informaciones se resumen en el estado "se ha introducido 1 euro".

Llamamos **estado actual** al estado en que se encuentra la máquina en un instante dado. El nuevo valor (moneda o ausencia de monedas) que llega por la entrada determinará cuál será el próximo estado en que se encontrará el circuito: el **estado futuro**.

Se llama **transición** al paso del estado actual a un estado futuro.



Una señal de reloj sincroniza los circuitos secuenciales. En cada flanco ascendente se produce una transición hacia un estado futuro u otro en función del valor de las entradas. Las consideraciones con respecto a la evolución temporal de los circuitos se tratan en otro subapartado.



La sincronización de los circuitos secuenciales se estudia con detalle en el subapartado 4.3. de este módulo.

Todas las transiciones que se pueden dar en un circuito secuencial se especifican mediante la **tabla de transiciones**.

La tabla de transiciones tiene a la izquierda todas las combinaciones posibles de estados actuales y valores de las entradas, y a la derecha, el estado futuro al que lleva cada combinación. A continuación se muestra la tabla de transiciones del ejemplo de la máquina de café (para hacer la tabla más legible, en lugar de escribir "se han introducido 0 euros" escribimos "cero", y de forma análoga para todos los estados).

| Estado actual | Entrada | Estado futuro |
|---|---|---|
| cero | Ninguna moneda | cero |
| cero | Moneda de 0,5 euros | medio |
| cero | Moneda de 1 euro | uno |
| medio | Ninguna moneda | medio |
| medio | Moneda de 0,5 euros | uno |
| medio | Moneda de 1 euro | uno y medio |
| uno | Ninguna moneda | uno |
| uno | Moneda de 0,5 euros | uno y medio |
| uno | Moneda de 1 euro | dos |
| uno y medio | Ninguna moneda | cero |
| uno y medio | Moneda de 0,5 euros | medio |
| uno y medio | Moneda de 1 euro | uno |
| dos | Ninguna moneda | cero |
| dos | Moneda de 0,5 euros | medio |
| dos | Moneda de 1 euro | uno |

**En cada transición, el estado futuro puede coincidir o no con el estado actual.**

**Nota**

Cuando ya se hayan introducido 1,5 euros o 2 euros, la máquina servirá un café y, si es el caso, dará el cambio. Si en una de estas situaciones se introduce otra moneda, el circuito entenderá que corresponde a una nueva petición de café.

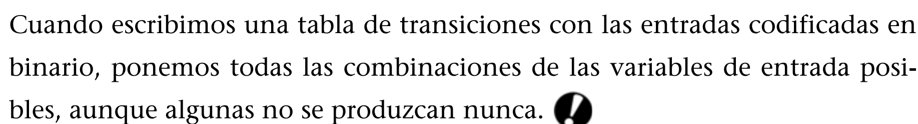
Un circuito secuencial siempre tiene un estado que refleja la situación "aún no ha pasado nada", es decir, "no es necesario recordar ninguno de los valores que ha llegado por la entrada". Este estado se denomina **estado inicial**.

Cuando un circuito se pone en funcionamiento, está en el estado inicial. Ahora bien, también puede estar en otros momentos, si la funcionalidad del circuito así lo requiere. En el caso de la máquina de café, el estado inicial es “se han introducido 0 euros”. Cuando la máquina haya servido un café y mientras no se introduzcan más monedas, volverá al estado inicial.

> Una vez especificado cuál es el estado inicial, la tabla de transiciones y la tabla de salidas describen completamente el comportamiento de un circuito lógico secuencial de acuerdo con el modelo de Moore.

La tabla de transiciones también se puede escribir con las señales de entrada codificadas en binario (ya que todo el circuito trabaja sólo con señales lógicas, como ya sabemos). En el ejemplo de la máquina de café, las entradas se pueden codificar en binario mediante las señales $m_1$ y $m_0$, tal como se ha visto en la tabla 1.

Cuando escribimos una tabla de transiciones con las entradas codificadas en binario, ponemos todas las combinaciones de las variables de entrada posibles, aunque algunas no se produzcan nunca.



En nuestro ejemplo, las variables $m_1$ y $m_0$ no tomarán nunca los valores [1 1]; pese a todo, ponemos estas combinaciones en la tabla de transiciones que se muestra a continuación. El valor del estado futuro en estos casos será *x*, ya que se trata de combinaciones *don't care*.

| Estado actual | $m_1$ | $m_0$ | Estado futuro |
|---|---|---|---|
| cero | 0 | 0 | cero |
| cero | 0 | 1 | medio |
| cero | 1 | 0 | uno |
| cero | 1 | 1 | × |
| medio | 0 | 0 | medio |
| medio | 0 | 1 | uno |
| medio | 1 | 0 | uno y medio |
| medio | 1 | 1 | × |
| uno | 0 | 0 | uno |
| uno | 0 | 1 | uno y medio |
| uno | 1 | 0 | dos |
| uno | 1 | 1 | × |
| uno y medio | 0 | 0 | cero |
| uno y medio | 0 | 1 | medio |
| uno y medio | 1 | 0 | uno |
| uno y medio | 1 | 1 | × |
| dos | 0 | 0 | cero |
| dos | 0 | 1 | medio |
| dos | 1 | 0 | uno |
| dos | 1 | 1 | × |

**Ejemplo de especificación de un circuito secuencial con el modelo Moore**

Veamos otro ejemplo de especificación de un circuito secuencial mediante el modelo de Moore.

Sea un circuito con una señal lógica de entrada, $x$, y una de salida, $z$. La señal de salida tiene que valer 1 siempre que se cumpla que por la entrada hayan llegado un número par de unos y un número par de ceros (recordemos que cero es un número par).

El circuito se puede encontrar en los siguientes estados:

- Ha llegado un número par de unos y un número par de ceros.
- Ha llegado un número par de unos y un número impar de ceros.
- Ha llegado un número impar de unos y un número par de ceros.
- Ha llegado un número impar de unos y un número impar de ceros.

El valor de la señal de salida en cada estado viene dado por la siguiente tabla de salidas:

| Estado<br>Ha llegado un número… | Salida<br>z |
|---|---|
| … par de unos y par de ceros | 1 |
| … par de unos e impar de ceros | 0 |
| … impar de unos y par de ceros | 0 |
| … impar de unos e impar de ceros | 0 |

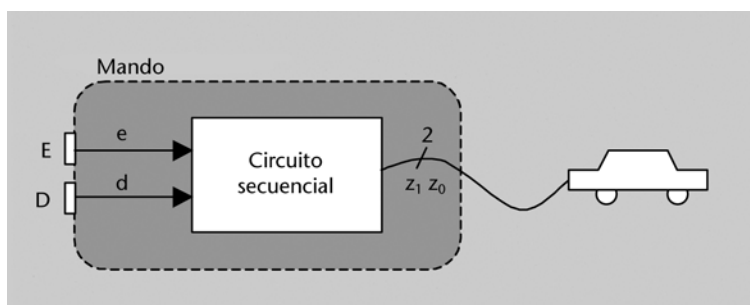
El circuito irá pasando por un estado u otro según si el valor de la entrada vale 0 ó 1. En concreto, la tabla de transiciones será la siguiente:

| Estado actual<br>Ha llegado un número… | Entrada<br>x | Estado futuro<br>Ha llegado un número… |
|---|---|---|
| … par de unos y par de ceros | 0 | … par de unos e impar de ceros |
| … par de unos y par de ceros | 1 | … impar de unos y par de ceros |
| … par de unos e impar de ceros | 0 | … par de unos y par de ceros |
| … par de unos e impar de ceros | 1 | … impar de unos e impar de ceros |
| … impar de unos y par de ceros | 0 | … impar de unos e impar de ceros |
| … impar de unos y par de ceros | 1 | … par de unos y par de ceros |
| … impar de unos e impar de ceros | 0 | … impar de unos y par de ceros |
| … impar de unos e impar de ceros | 1 | … par de unos e impar de ceros |

**Para referirnos a *estado actual* y *estado futuro*, también podemos escribir Estado y Estado$^+$.**

## Actividades

**21.** Se quiere diseñar el mando de control remoto de un coche de juguete. El mando tiene dos botones: *E* y *D*.




Si el coche está parado, al pulsar cualquier botón se pone en movimiento: gira a la izquierda si se pulsa *E*, gira a la derecha si se pulsa *D* y va adelante si se pulsan los dos botones a la vez. Mientras el coche está en movimiento, si se pulsa *E* hará lo siguiente:

- girará a la izquierda si iba recto,
- irá recto si giraba a la derecha,
- continuará girando a la izquierda si ya lo hacía.

Sucederá de forma análoga cuando se pulse *D*. Cuando se pulsen los dos botones a la vez, se parará.

El mando dispondrá de un circuito secuencial que recibe como entrada dos señales *e* y *d* conectadas a los botones *E* y *D* respectivamente (1: pulsar; 0: no pulsar), y genera dos señales, $z_1$ y $z_0$, que gobernarán el coche según la tabla que vemos al margen.

| $z_1$ | $z_0$ | Acción del coche |
|---|---|---|
| 0 | 0 | Girar a la derecha |
| 0 | 1 | Girar a la izquierda |
| 1 | 0 | Detenerse |
| 1 | 1 | Moverse adelante |

**a)** ¿Qué estados tiene el circuito?
**b)** ¿Cuál es el estado inicial?
**c)** Escribid la tabla de salidas y la de transiciones.

**22.** Sea un circuito secuencial con dos señales de entrada *x* e *y*, ambas de un bit, y una señal de salida *z* también de un bit. La señal de salida se tiene que poner a 1 cuando en al menos tres ocasiones los valores de *x* e *y* se hayan igualado.

**a)** ¿Qué estados tiene el circuito?
**b)** ¿Cuál es el estado inicial?
**c)** Escribid la tabla de salidas y la de transiciones.

### 4.2. Representación gráfica: grafos de estado

La especificación del funcionamiento de un sistema secuencial mediante el modelo de Moore se representa gráficamente con un **grafo de estados**, de la forma siguiente:

**1)** Para cada estado se dibuja un círculo, con el nombre del estado en la parte superior. El círculo correspondiente al estado inicial se señala con una flecha y la palabra *Inicio*.

**Recordemos**

Los circuitos secuenciales suelen tener una señal de entrada *Inicio* que genera un pulso a 1 para indicar al circuito que se ponga en funcionamiento.

**2)** En la parte inferior de cada círculo se escribe el valor que toman las señales de salida cuando el circuito se encuentra en el estado correspondiente a este círculo.

**3)** Las transiciones se representan mediante flechas o **arcos**, que tienen su origen en el círculo correspondiente al estado actual y la punta en el círculo correspondiente al estado futuro. El valor de las señales de entrada asociado con la transición se escribe al lado del arco.

**Los valores escritos al lado de un arco también se llaman *etiquetas* del arco.**

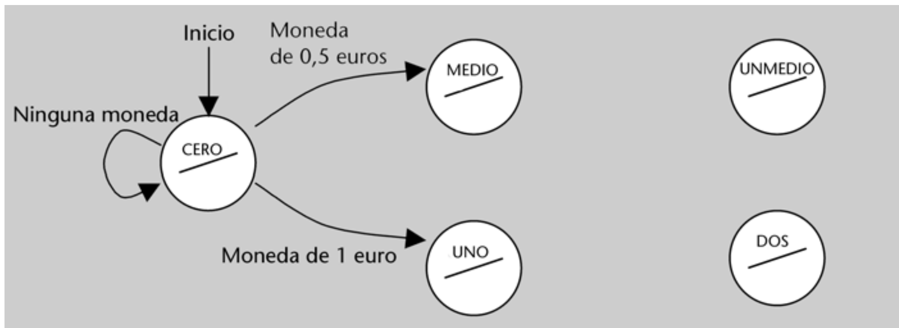
La figura 15 muestra la representación gráfica de los estados de la máquina de café, y las transiciones que parten del estado "se han introducido 0 euros". En esta figura hemos dado a los diferentes estados estos nombres:

| Estado | Nombre |
|---|---|
| Se han introducido 0 euros. | CERO |
| Se han introducido 0,5 euros. | MEDIO |
| Se ha introducido 1 euro. | UNO |
| Se han introducido 1,5 euros. | UNMEDIO |
| Se han introducido 2 euros. | DOS |

**Consejo práctico**

Al especificar un circuito secuencial mediante un grafo de estados, se suele dar a cada estado un nombre corto, ya que así se puede escribir cómodamente. El nombre que se dé a cada estado es indiferente, pero resulta práctico que sea un mnemotécnico que nos remita de alguna forma a la situación que refleja cada estado.

Figura 15



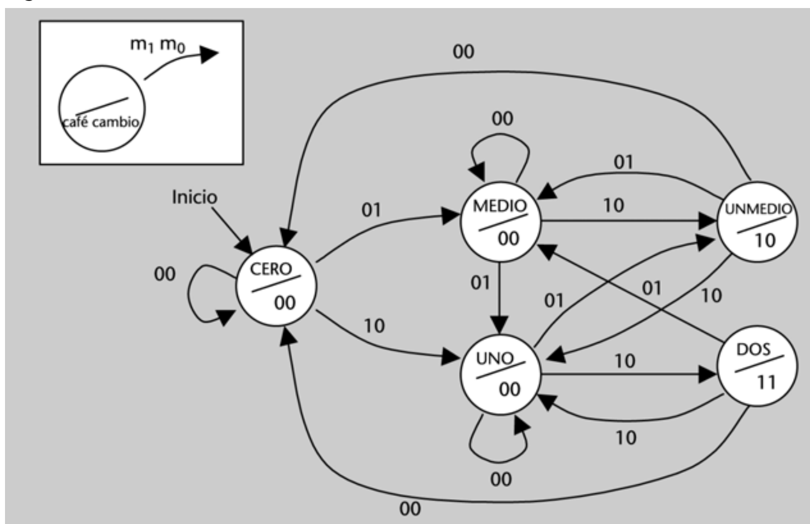

El grafo completo se muestra en la figura 16, con las entradas codificadas en binario (de acuerdo con la tabla 1). En el recuadro de la parte superior izquierda encontramos la leyenda del grafo.

La **leyenda** del grafo de estados indica el orden en que se escriben las señales de entrada en la etiquetas de los arcos y las señales de salida dentro de los círculos.

**La leyenda de un grafo**

Si un grafo con más de una señal de entrada o de salida no dispone de leyenda, es imposible descifrar su significado.

Figura 16




**Combinaciones imposibles**

En general, en un grafo no aparecen las combinaciones de las variables de entrada que no se darán nunca. Por este motivo, en la figura 16 no hay ninguna etiqueta [1 1].

Gracias a la leyenda sabemos que, en las parejas de valores de las etiquetas de la figura 16, el de la izquierda corresponde a $m_1$ y el de la derecha a $m_0$ (y no al revés). La leyenda nos indica también que, de los valores de salida, el de la izquierda corresponde a la señal *café* y el de la derecha, a la señal *cambio*.

### 4.2.1. Mecánica de diseño

A partir del enunciado del comportamiento de un circuito secuencial, para encontrar el grafo de estados podemos seguir al algoritmo que presentamos a continuación:

**1)** Analizar qué entradas y salidas tiene el circuito y determinar la leyenda del grafo.

**2)** Dibujar un círculo para el estado inicial, darle un nombre y escribir el valor de las salidas en este estado.

**3)** Hacer una lista de todas las combinaciones de valores que pueden tomar las señales de entrada en este estado. Para cada una, deducir qué transición provoca. Si la transición comporta la aparición de un estado inexistente hasta el momento, incorporarlo al grafo, darle un nombre y escribir el valor adecuado para las salidas. Dibujar el arco correspondiente a la transición.

**4)** Repetir el paso 3 para todos los estados nuevos que hayan aparecido, hasta que no aparezca ninguno nuevo.

**Circuitos reconocedores de secuencia**

Veamos un ejemplo de construcción del grafo a partir de la especificación del funcionamiento del circuito.

Se quiere dibujar el grafo de estados de un circuito con una señal de entrada *x* y una de salida *z*, ambas de un bit. Inicialmente, la salida tiene que valer 0. Cuando en la entrada se haya producido la secuencia de valores 101, la salida se tiene que poner a 1. La salida se tiene que volver a poner a 0 cuando por la entrada haya llegado la secuencia de valores 001.

Los circuitos que generan un 1 en la salida cuando se ha producido una secuencia de valores determinada en la entrada se llaman *reconocedores de secuencia.*

En este caso, las señales de entrada y de salida son de un único bit; por tanto, no hay que determinar la orden de las señales en la leyenda, sólo podemos poner el nombre.

El circuito tendrá un estado inicial, llamado *INI* con salida 0. Puesto que la entrada es de un único bit, sólo puede tomar los valores 0 y 1.

Si estando en el estado *INI*, por la entrada llega un 1, tenemos que recordarlo, ya que puede ser el principio de la secuencia 101 que queremos reconocer. Aparece, pues, un nuevo estado, que llamamos "ha llegado un 1"; la salida en este estado continúa valiendo 0. En cambio, si estando en el estado inicial llega un 0, no es preciso recordarlo, y nos quedaremos en el mismo estado *INI*.

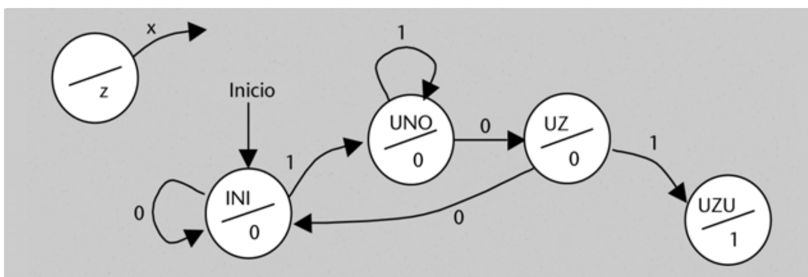
Estando en el estado "ha llegado un 1", si llega un 0 tenemos que pasar a un estado nuevo, que llamaremos "ha llegado la subsecuencia 10", con salida 0. Si llega un 1, se rompe la secuencia anterior, pero este nuevo 1 puede ser a su vez el inicio de una nueva secuencia 101. Por tanto, nos tenemos que quedar en el mismo estado "ha llegado un 1".

Situémonos ahora en el nuevo estado que ha aparecido, "ha llegado la subsecuencia 10". Si llega un 0, tenemos que los tres últimos valores que han llegado por la entrada son 100 y, por tanto, ninguno de ellos puede formar parte de la secuencia 101. Volvemos, pues, al estado *INI*. En cambio, si llega un 1, tenemos que los tres últimos valores que han llegado son 101, que es justamente la secuencia que el circuito tiene que reconocer. Pasaremos, pues, a un nuevo estado "ha llegado la secuencia 101", en el que la salida vale 1.

Demos a los estados que han aparecido hasta ahora los siguientes nombres:

| Estado | Nombre |
|---|---|
| Ha llegado un 1 | UNO |
| Ha llegado la subsecuencia 10 | UZ |
| Ha llegado la secuencia 101 | UZU |

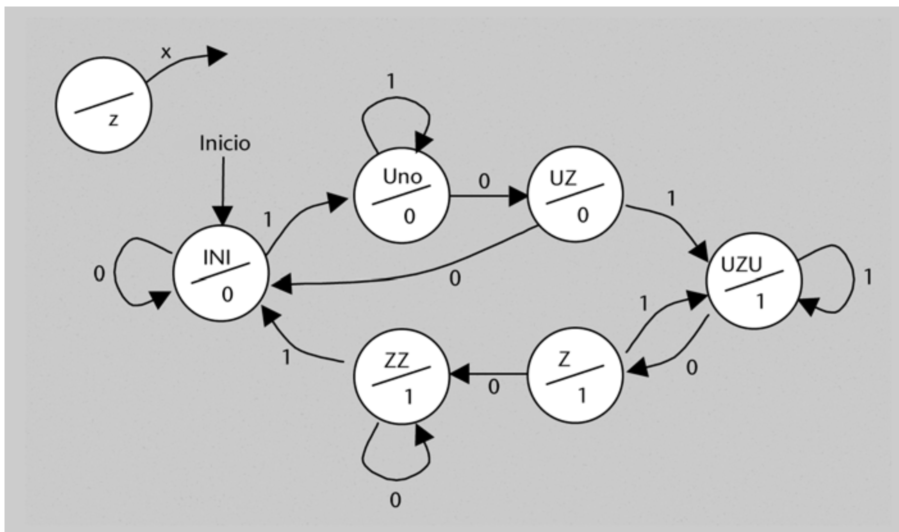
La parte del grafo que hemos construido hasta ahora se muestra a continuación:



Una vez ha llegado al estado *UZU*, el circuito tiene que reconocer la secuencia 001 para saber cuándo tiene que volver a poner la salida a 0. Razonemos de forma análoga al caso anterior y obtendremos que deberá presentar los siguientes estados:

| Estado | Nombre |
|---|---|
| Ha llegado un 0 | Z |
| Ha llegado la subsecuencia 00 | ZZ |
| Ha llegado la secuencia 001 | ZZU |

Sin embargo, vemos que el estado ZZU coincide con el estado inicial, ya que la salida debe valer 0 y el circuito tiene que reconocer a partir de este momento la secuencia 101; es decir, el circuito se encuentra en la misma situación que al empezar a funcionar. Por último, obtenemos el siguiente grafo completo:

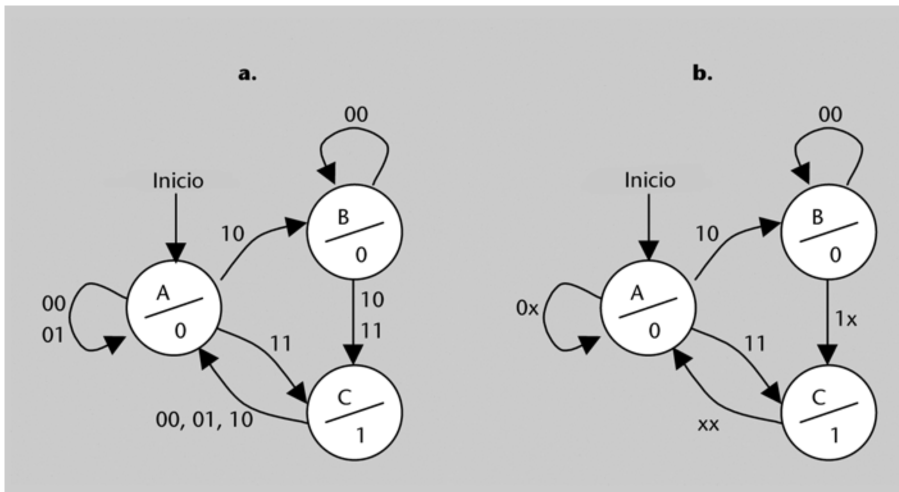



### 4.2.2. Notación

En un grafo, si se produce una transición de un estado determinado hacia el mismo estado futuro por más de una combinación de valores de las señales de entrada, escribiremos las etiquetas correspondientes una debajo de otra, o bien separadas por comas, tal como se muestra en el gráfico **a** de la figura 17. Observamos que, en el circuito correspondiente a este grafo, estando en el estado *B* no se dará nunca la combinación de entrada [0 1], y estando en el estado *C* no se dará nunca la combinación [1 1].

En una etiqueta podemos escribir también *x* para referirnos a un valor cualquiera de una señal de entrada (igual que se hace en las tablas de la verdad). Cuando de un estado se pasa siempre a un mismo estado de futuro, independientemente del valor de las entradas (como es el caso del estado *C* en el grafo **a** de la figura 17), se puede poner una sola etiqueta *x* como valor de todas las variables, incluso si algunas combinaciones no se dan nunca. Así, los dos grafos de la figura 17 son equivalentes (observemos que, estando en el estado *C*, la combinación de entrada 11 no se puede producir).

Figura 17



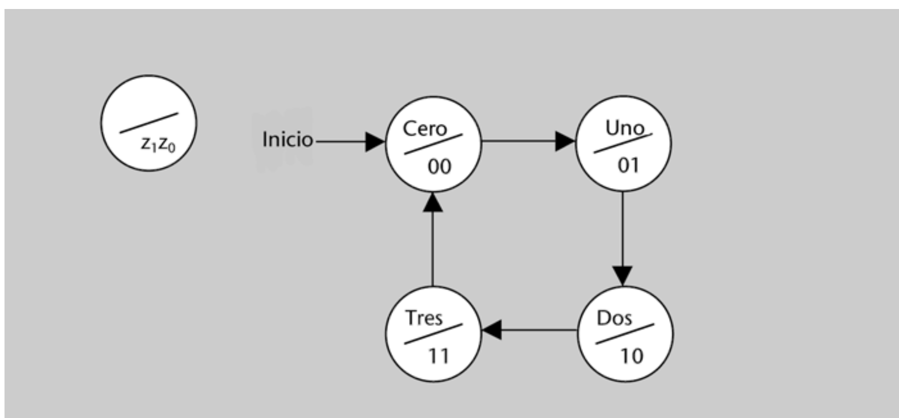

### 4.2.3. Circuitos sin entradas

Un circuito secuencial puede no tener ninguna señal. En este caso, siempre se producirán las mismas transiciones entre estados y, por tanto, la secuencia de valores en las salidas será siempre la misma.

**Los contadores módulo *n***

En general, un contador módulo *n* es un circuito que genera cíclicamente la secuencia de valores 0, 1, ..., *n* – 1.

Imaginemos un circuito cuya misión es generar cíclicamente la secuencia de números 0, 1, 2 y 3, codificados en binario. Este circuito recibe el nombre de *contador módulo 4*, y su grafo de estados es el que se muestra en la siguiente figura.




**Nota**

Observad que cuando el circuito no tiene ninguna entrada, al diseñar su grafo de estados, la leyenda no tiene ningún arco ni ninguna etiqueta asociada al mismo.

## Actividades

**23.** Dibujad el grafo de estados del circuito que se describe en la actividad 17.

**24.** Dibujad el grafo de estados del circuito secuencial que se comporta tal como describen las siguientes tablas:

**a)** Estado inicial: *A*

| Tabla de transiciones | | | |
|---|---|---|---|
| Estado | $x_1$ | $x_0$ | Estado⁺ |
| A | 0 | 0 | A |
| A | 0 | 1 | C |
| A | 1 | 0 | B |
| A | 1 | 1 | B |
| B | 0 | 0 | B |
| B | 0 | 1 | D |
| B | 1 | 0 | A |
| B | 1 | 1 | A |
| C | 0 | 0 | A |
| C | 0 | 1 | C |
| C | 1 | 0 | D |
| C | 1 | 1 | D |
| D | 0 | 0 | D |
| D | 0 | 1 | B |
| D | 1 | 0 | C |
| D | 1 | 1 | C |

| Tabla de salidas | | | |
|---|---|---|---|
| Estado | $y_2$ | $y_1$ | $y_0$ |
| A | 1 | 0 | 0 |
| B | 0 | 1 | 0 |
| C | 0 | 0 | 1 |
| D | 1 | 1 | 1 |

**b)** Estado inicial: *C*

| Tabla de transiciones | | | |
|---|---|---|---|
| Estado | $e_1$ | $e_0$ | Estado⁺ |
| A | 0 | x | B |
| A | 1 | 0 | C |
| A | 1 | 1 | A |
| B | 0 | 0 | B |
| B | 0 | 1 | C |
| B | 1 | 0 | A |
| B | 1 | 1 | x |
| C | 0 | x | A |
| C | 1 | 0 | A |
| C | 1 | 1 | x |

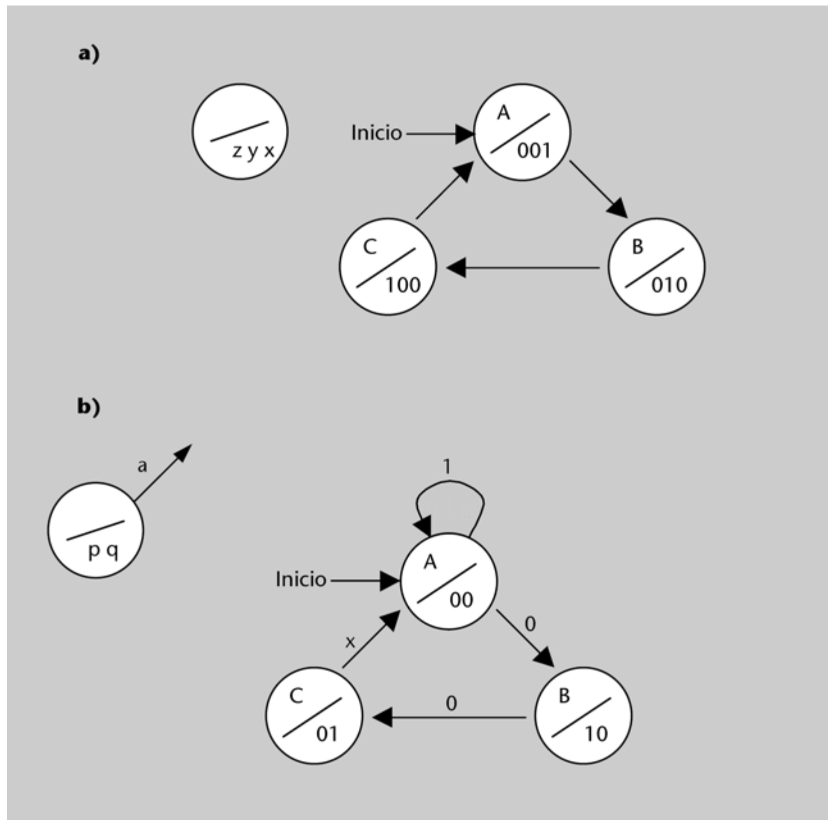
| Tabla de salidas | |
|---|---|
| Estado | z |
| A | 1 |
| B | 1 |
| C | 0 |

**c)** Estado inicial: *E2*

| Tabla de transiciones | |
|---|---|
| Estado | Estado⁺ |
| E0 | E2 |
| E1 | E0 |
| E2 | E1 |

| Tabla de salidas | |
|---|---|
| Estado | z |
| E0 | 1 |
| E1 | 0 |
| E2 | 1 |

**25.** Escribid las tablas de transiciones y salidas de los circuitos que se comportan tal como describen los grafos de estados siguientes:



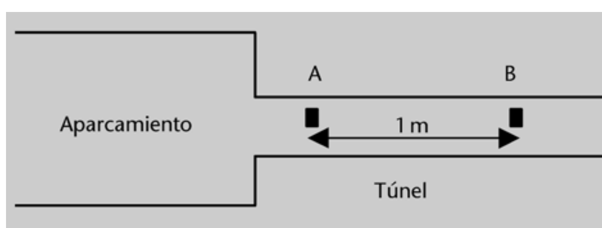

**26.** Dibujad el grafo de estados de un circuito que funcione como un contador reversible módulo 5. Un contador reversible cuenta adelante o atrás en función de una señal de entrada $x$:

- $x = 0$: cuenta adelante
- $x = 1$: cuenta atrás

Inicialmente, la salida tiene que valer 0. La salida no se codificará en sistema binario, sino como se indica en esta tabla:

| Valor de la salida | Señales de salida $z_3z_2z_1z_0$ |
|---|---|
| 0 | 0 0 0 0 |
| 1 | 0 0 0 1 |
| 2 | 0 0 1 1 |
| 3 | 0 1 1 1 |
| 4 | 1 1 1 1 |

**27.** En un aparcamiento se necesita saber el número de coches que hay en cada momento. Los coches entran y salen por el mismo túnel, en el que sólo cabe un coche. En el túnel hay dos sensores, *A* y *B*, separados por un metro, de forma que se puede saber si un coche entra o sale según el orden en que se activen los sensores (se supone que todos los coches miden más de un metro y que entre un coche y el siguiente habrá más de un metro). Dos coches no es encontrarán nunca de cara en el túnel. Tampoco habrá peatones.



Cuando un coche ha entrado totalmente en el aparcamiento se incrementa el número de coches aparcados, y cuando ha salido totalmente se decrementa. Mientras pasa por el túnel, un coche puede parar o hacer marcha atrás en cualquier momento.

Dibujad el grafo de estados de un circuito secuencial que a partir de las señales *a* y *b*, provenientes respectivamente de los sensores *A* y *B* (valdrán 1 si hay un coche ante un sensor y 0 si no hay ninguno), genere dos señales de salida *más* o *menos* que gobernarán el contador de coches que hay en el aparcamiento en cada momento.

## 4.3. Sincronización

Ya sabemos que los circuitos secuenciales están sincronizados por una señal de reloj que describe ciclos periódicos entre los valores 0 y 1.

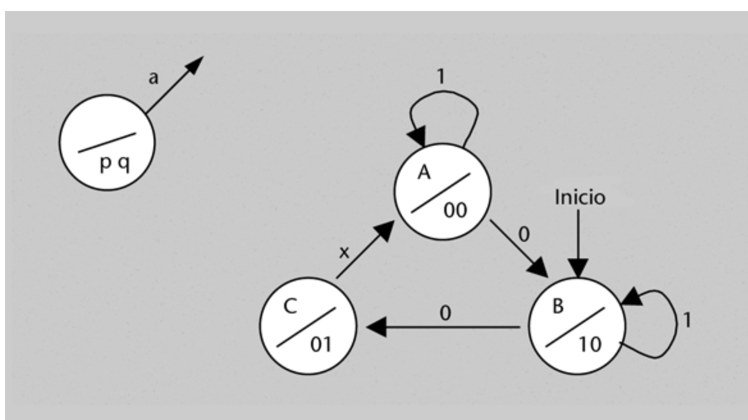
Las transiciones entre estados tienen lugar en cada flanco ascendente de reloj (porque los estados se implementan físicamente mediante biestables).

Observad la implementación de los estados mediante biestables en el subapartado 4.4. de este módulo.

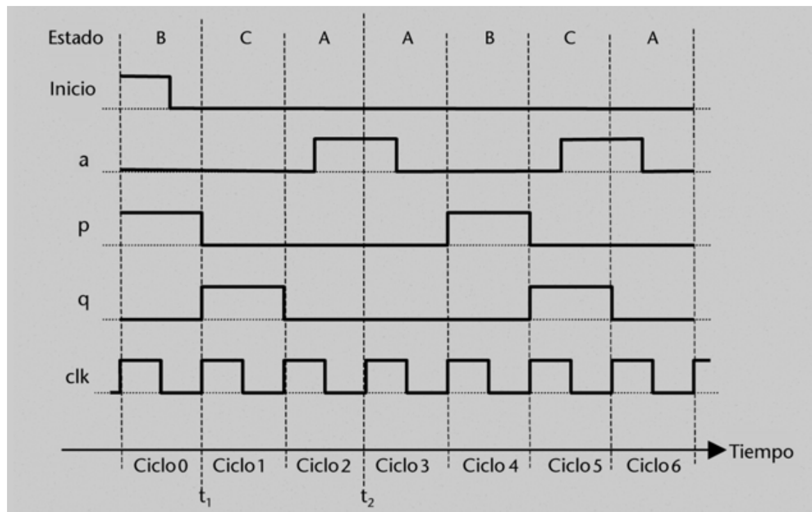
Por tanto, el circuito examina el valor de las señales de entrada en cada ciclo de reloj. En concreto, el valor que hace que se tome una transición u otra es lo que tienen las entradas al llegar al instante del flanco. Si dibujamos las entradas en un cronograma, el valor que decide qué transición se toma es el que tienen al tocar, por la izquierda, la línea vertical correspondiente a un flanco.

**Ejemplo de transiciones entre estados**

Sea el grafo de estados que se muestra en esta figura:




La siguiente figura muestra cómo evoluciona el circuito con el tiempo a partir de una secuencia de valores determinada en la entrada *a*. En el ciclo 0, la señal *Inicio* genera un pulso a 1 y hace que el circuito se ponga en el estado *B*. En el instante $t_1$ (por la izquierda), la señal de entrada *a* vale 0, lo que provoca que el circuito pase al estado *C* en ese instante. Durante el ciclo 2, el circuito se encuentra en el estado *A*. Dado que el instante $t_2$ la entrada vale 1, en este momento se produce una transición hacia el mismo estado *A*. El resto del cronograma se calcula de forma análoga.

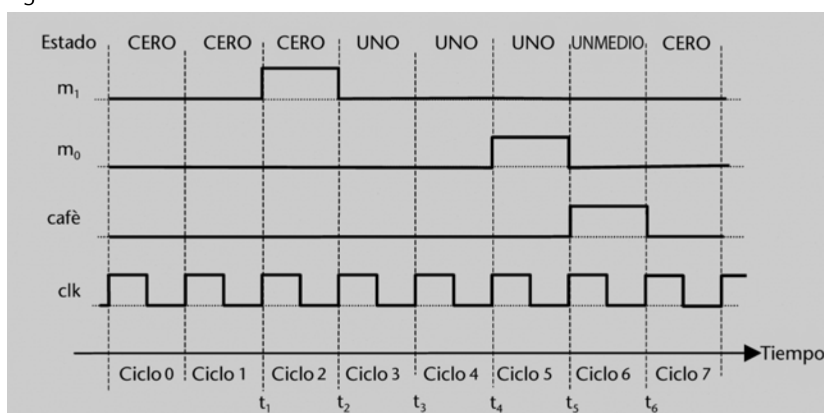



Recordemos que el valor de las señales de salida en cada momento viene determinada por el estado en que se encuentra el circuito.

Retomemos el ejemplo de la máquina de café. Cuando se introduce una moneda, pasa un intervalo de tiempo desde que se introduce en la ranura hasta que cae en la caja correspondiente. Supongamos que el sistema que codifica las señales $m_1$ y $m_0$ (recordemos la tabla 1 y la figura 14) genera pulsos de un ciclo de duración: $m_0$ estará a 1 durante un ciclo cuando se haya introducido una moneda de 0,5 euros, y $m_1$ estará a 1 durante un ciclo cuando se haya introducido una moneda de 1 euro.

La figura 18 muestra una posible evolución temporal del circuito, partiendo del estado *CERO*. Durante los dos primeros ciclos no se ha introducido ninguna moneda y, por tanto, las transiciones que se han producido han llevado siempre al estado *CERO*. En el ciclo 2, $m_1$ genera un pulso a 1 e indica que se ha introducido una moneda de 1 euro. Por tanto, el circuito pasa al estado *UNO* en el instante $t_2$. Las próximas dos transiciones llevarán también al estado *UNO*, ya que $[m_1\ m_0]$ = [0 0] en los instantes $t_3$ y $t_4$. En el ciclo 5, $m_0$ = 1 (se ha introducido una moneda de 0,5 euros) y, por tanto, el circuito pasa al estado *UNMEDIO* en el instante $t_5$. Por tanto, durante el ciclo 6, la salida *café* vale 1 (eso hará que se active el dispositivo que sirve un café). Puesto que durante este ciclo las entradas valen 0 (no se ha introducido ninguna moneda), el circuito pasa al estado *CERO* en el instante $t_6$ y, por tanto, la salida *café* vuelve a 0.

Figura 18




**Representación de los ciclos en un cronograma**

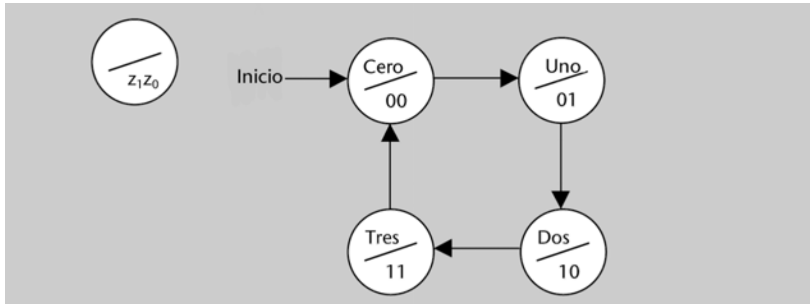
En este cronograma no hemos dibujado la señal *Inicio*. En general, los ciclos que mostramos en un cronograma no tienen por qué ser iniciales, sino que pueden corresponder a un momento cualquiera del funcionamiento del circuito.

Vemos pues que, en los circuitos secuenciales, el tiempo que el circuito esté en cada estado y, por tanto, la duración de los diferentes valores de las señales de salida, vienen determinados por la sincronización.

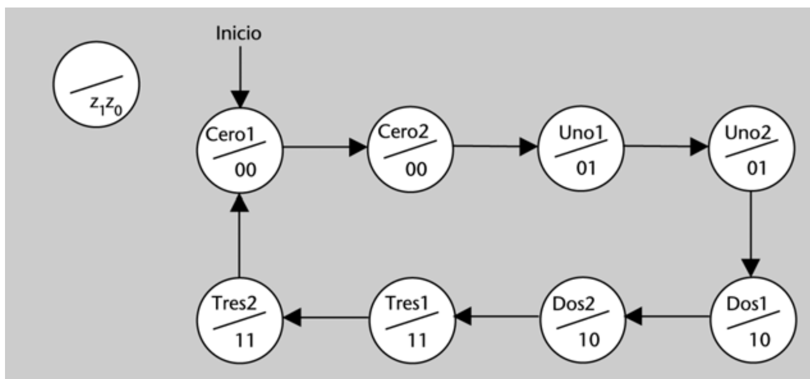
**Un contador módulo 4**

Queremos diseñar un contador módulo 4 como el de la siguiente figura en el que cada valor de salida dure 100 ns:

Ved el ejemplo de los contadores módulo *n* en el subapartado 4.2.3



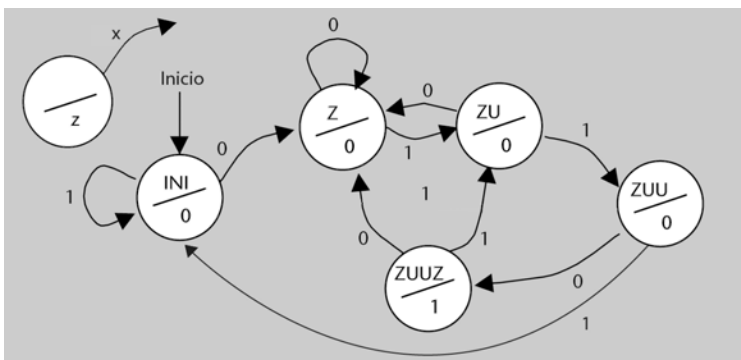

Si disponemos de una señal de reloj con un periodo de 100 ns, entonces el grafo de estados del circuito es el que se muestra en la figura anterior. Sin embargo, si el periodo del reloj es de 50 ns, entonces el grafo tiene que ser el siguiente:



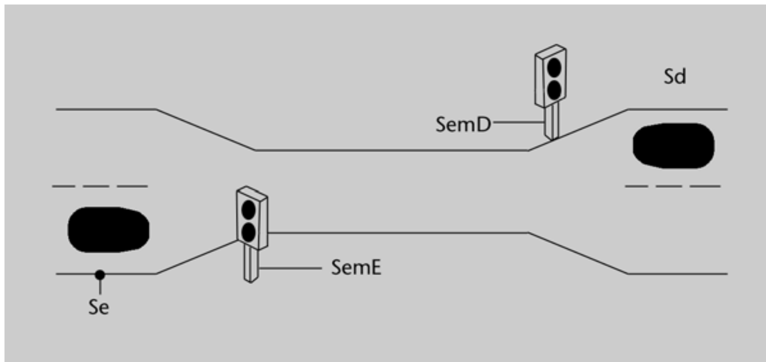

## Actividades

**28.** Dibujad el grafo de estados de un circuito que reconozca la secuencia 0110 en la entrada *x* (de un bit). Al reconocerla, la señal de salida *z* (de un bit) se tiene que poner a 1 durante un ciclo de reloj. Inicialmente la salida tiene que estar a 0.
El circuito no detecta solapamiento entre dos secuencias consecutivas. Es decir, si llegan los valores de entrada 0110110, la salida sólo se pondrá a uno después de los cuatro primeros valores.

**29.** El grafo de estados siguiente corresponde a un circuito que reconoce una secuencia determinada de valores en la señal de entrada *x* y pone la salida *z* a 1 durante un ciclo cuando se ha producido. ¿Cuál es esta secuencia? Describid con detalle los casos en que se produce el reconocimiento.



**30.** Dibujad el grafo de estados de un circuito secuencial que controle los semáforos de un puente en el que no hay suficiente espacio para que circulen simultáneamente coches en ambos sentidos.



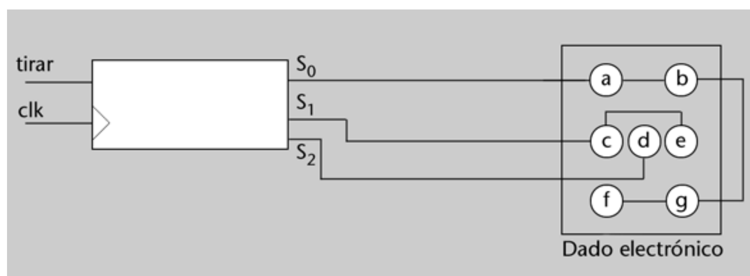

En cada extremo del puente hay:

- un sensor que pone la señal *Sd* (o *Se*) a 1 cuando hay un coche delante.
- un semáforo que se pone en rojo cuando le llega un 0 por la señal *SemD* (o *SemE*) y se pone en verde cuando le llega un 1.

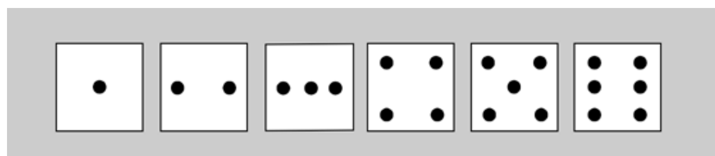
El circuito recibe como entrada las señales *Sd* y *Se* y genera como salida las señales *SemD* y *SemE*. Debe funcionar de la siguiente forma:

- Los semáforos se ponen en verde de forma alternada durante dos ciclos de reloj como mínimo.
- Después del segundo ciclo de reloj, el semáforo que está en verde permanece así hasta que llegue un coche por el otro extremo.
- Al ponerse el sistema en funcionamiento, el semáforo de la izquierda tiene que estar en verde durante dos ciclos.

**31.** Se quiere diseñar un circuito secuencial para tirar un dado. La entrada de la señal *tirar*, que está conectada a un pulsador que manipula el jugador (pulsado: *tirar* = 1; no pulsado: *tirar* = 0). Las salidas del circuito estarán conectadas a los puntos de un "dado electrónico" de la siguiente forma:



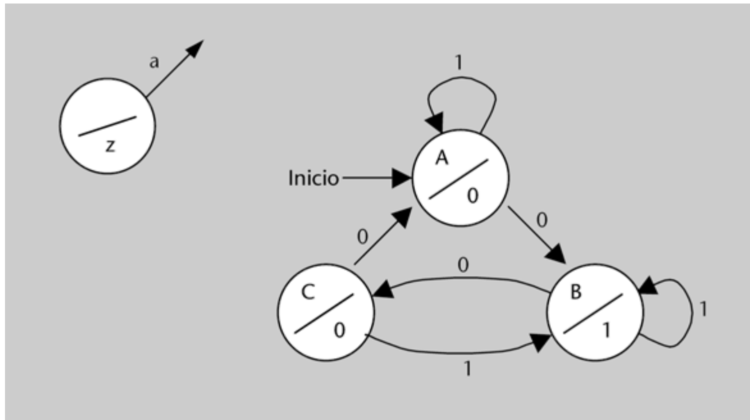

Los puntos se iluminan cuando les llega un 1. Las combinaciones posibles del dado son las siguientes:



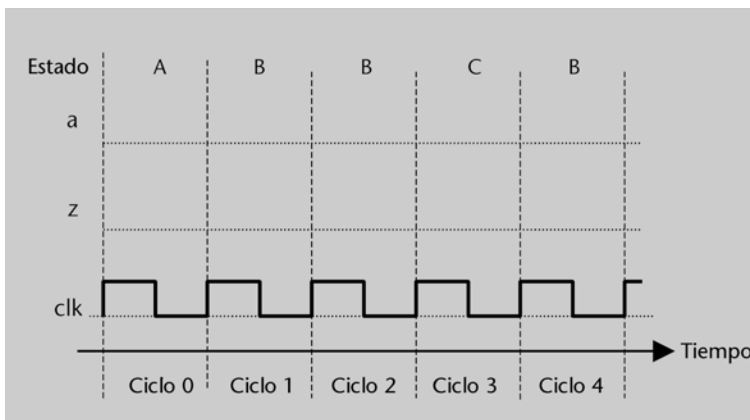
Cuando el dado empiece a funcionar, tiene que mostrar un "1". A partir del momento en que el jugador pulse el pulsador y mientras lo mantenga pulsado, el circuito generará cíclicamente y en orden las seis combinaciones del dado. Cuando el jugador libere el pulsador, no se producirá ninguna transición, y verá la combinación que ha salido en el dado (se supone que la frecuencia del reloj es muy alta y no se pueden llegar a distinguir las combinaciones intermedias).

Dibujad el grado de estados del circuito.

**32.** Sea el siguiente grado de estados:



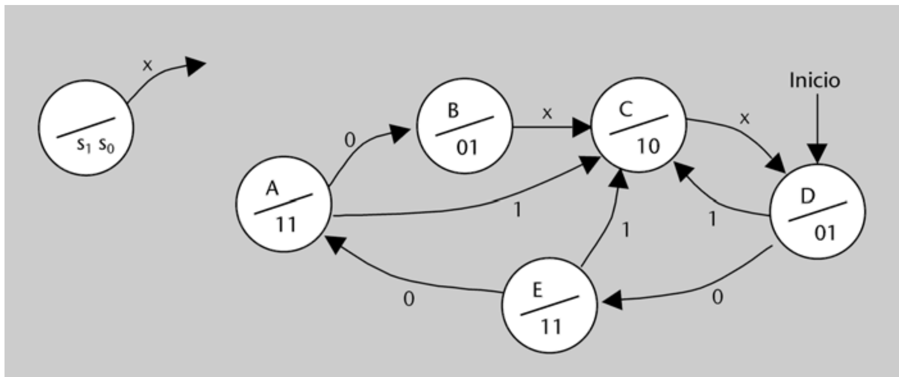

Completad el siguiente cronograma. Suponed que la señal de entrada *a* sólo cambia de valor en los instantes de los flancos.




### 4.4. Implementación

Veamos cómo se puede implementar físicamente un circuito secuencial descrito de acuerdo con el modelo de Moore. Tomemos como ejemplo el circuito descrito por el grafo de la figura siguiente:

Figura 19



A continuación se muestran las tablas de transiciones y de salidas correspondientes a este grafo:

| Tabla de transiciones | | |
|---|---|---|
| Estado | x | $Estado^+$ |
| A | 0 | B |
| A | 1 | C |
| B | 0 | C |
| B | 1 | C |
| C | 0 | D |
| C | 1 | D |
| D | 0 | E |
| D | 1 | C |
| E | 0 | A |
| E | 1 | C |

| Tabla de salidas | | |
|---|---|---|
| Estado | $s_1$ | $s_0$ |
| A | 1 | 1 |
| B | 0 | 1 |
| C | 1 | 0 |
| D | 0 | 1 |
| E | 1 | 1 |

Estas dos tablas se pueden convertir fácilmente en tablas de la verdad de funciones lógicas si codificamos los estados mediante variables lógicas. En concreto, si hay *n* estados, serán necesarios $\lceil \log_2 n \rceil$ variables para codificarlos. En nuestro ejemplo, ésta es una posible codificación de estados:

| Estado | $q_2$ | $q_1$ | $q_0$ |
|---|---|---|---|
| A | 0 | 0 | 0 |
| B | 0 | 0 | 1 |
| C | 0 | 1 | 0 |
| D | 0 | 1 | 1 |
| E | 1 | 0 | 0 |

Una vez codificados los estados, las transiciones y salidas del circuito son funciones lógicas que se describen con las siguientes tablas de la verdad:

| Tabla de transiciones | | | | | | |
|---|---|---|---|---|---|---|
| Estado | | | | $Estado^+$ | | |
| $q_2$ | $q_1$ | $q_0$ | x | $q_2^+$ | $q_1^+$ | $q_0^+$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | x | x | x |
| 1 | 0 | 1 | 1 | x | x | x |
| 1 | 1 | 0 | 0 | x | x | x |
| 1 | 1 | 0 | 1 | x | x | x |
| 1 | 1 | 1 | 0 | x | x | x |
| 1 | 1 | 1 | 1 | x | x | x |

| Tabla de salidas | | | | |
|---|---|---|---|---|
| Estado | | | | |
| $q_2$ | $q_1$ | $q_0$ | $s_1$ | $s_0$ |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | x | x |
| 1 | 1 | 0 | x | x |
| 1 | 1 | 1 | x | x |

Las variables que codifican los estados, $q_i$, se guardan en biestables. De esta forma, el circuito guarda en todo momento la memoria del estado en que se encuentra. En nuestro ejemplo, serán necesarios tres biestables; cuando éstos valgan, por ejemplo, $[q_2\ q_1\ q_0] = [0\ 1\ 0]$, sabremos que el circuito se encuentra en el estado *C*.

Las señales de salida $s_1$ y $s_0$ se pueden implementar como funciones lógicas de $q_2$, $q_1$ y $q_0$, a partir de la tabla de verdad anterior, de cualquiera de las formas que conocéis.

Podéis ver varias formas de implementar funciones lógicas en el módulo "Los circuitos lógicos combinacionales" de esta asignatura.

Por lo que se refiere a las transiciones, las columnas $q_2^+$, $q_1^+$ y $q_0^+$ nos indican los valores que deben tomar los biestables en el siguiente flanco de reloj. Dado que un biestable toma el valor que hay en su entrada *D*, sabemos que en las entradas de los biestables tenemos que poner, para cada una de las combinaciones posibles de estados y entradas, lo que muestra la siguiente tabla (en la que $d_i$ corresponde a la entrada *D* de cada uno de los biestables):

| $q_2$ | $q_1$ | $q_0$ | x | $d_2$ | $d_1$ | $d_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | x | x | x |
| 1 | 0 | 1 | 1 | x | x | x |
| 1 | 1 | 0 | 0 | x | x | x |
| 1 | 1 | 0 | 1 | x | x | x |
| 1 | 1 | 1 | 0 | x | x | x |
| 1 | 1 | 1 | 1 | x | x | x |

Esta tabla se llama **tabla de excitaciones**, ya que nos indica cómo hay que "excitar" los biestables para que tengan lugar las transiciones adecuadas. Fijémonos en que las columnas $d_i$ de la tabla de excitaciones coinciden con las columna $q_i^+$ de la tabla de transiciones.

Las señales $d_2$, $d_1$ y $d_0$, llamadas **funciones de excitación**, son funciones lógicas de $q_2$, $q_1$ y $q_0$, y de la entrada *x*. Podemos implementarlas, pues, por cualquiera de los métodos que conocemos.

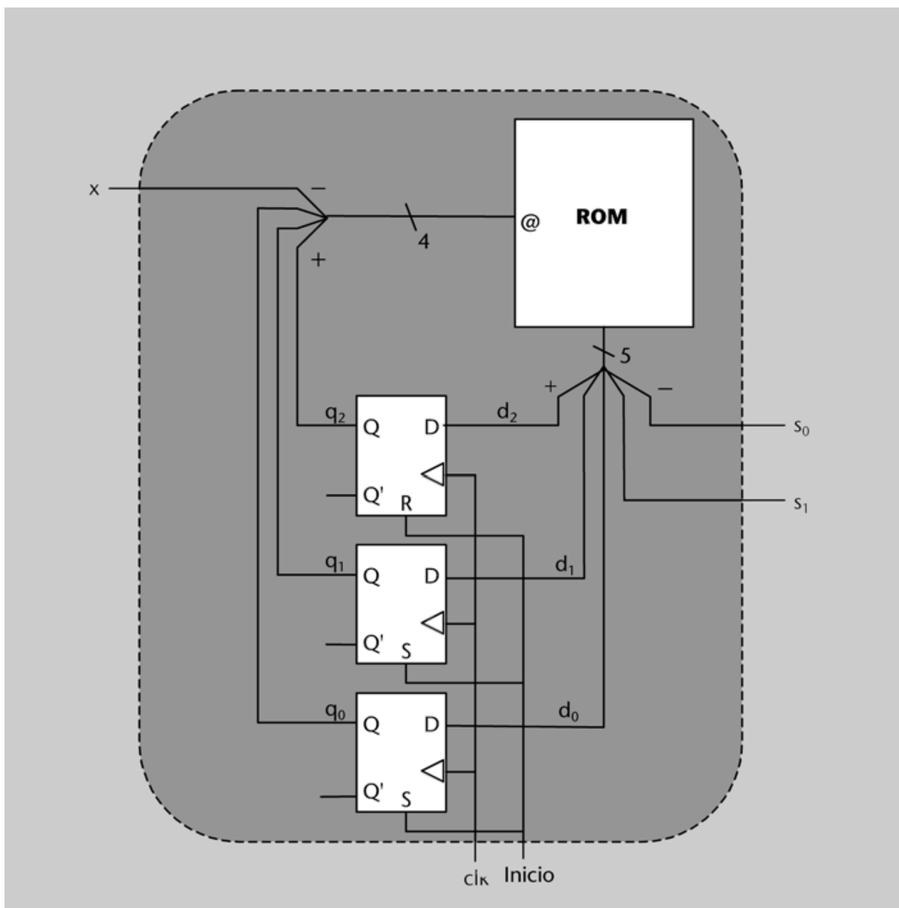
Podemos escribir una sola tabla de verdad que incluya las funciones de excitación y las de salida, tal como se muestra en la tabla siguiente. Fijémonos en que, por el hecho de que las funciones de salida dependen sólo del estado (va-

riables $q_i$), su valor es el mismo en las filas correspondientes a una misma combinación de $q_i$, y a diferentes valores de la entrada $x$.

| $q_2$ | $q_1$ | $q_0$ | x | $d_2$ | $d_1$ | $d_0$ | $s_1$ | $s_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | x | x | x | x | x |
| 1 | 0 | 1 | 1 | x | x | x | x | x |
| 1 | 1 | 0 | 0 | x | x | x | x | x |
| 1 | 1 | 0 | 1 | x | x | x | x | x |
| 1 | 1 | 1 | 0 | x | x | x | x | x |
| 1 | 1 | 1 | 1 | x | x | x | x | x |

Una vez se ha expresado el comportamiento del circuito mediante esta única tabla de verdad, se puede implementar de forma muy sencilla utilizando una memoria ROM, tal como se muestra en la figura que presentamos a continuación:

Figura 20




**Nota**

En este circuito los biestables están dibujados con las entradas a la derecha y las salidas a la izquierda.

**La memoria ROM de la figura 20 se podría sustituir por cualquier otra forma de implementación de funciones lógicas.**

El contenido de la memoria ROM, que corresponde a la tabla anterior, es el siguiente:

| Dirección | $d_2$ | $d_1$ | $d_0$ | $s_1$ | $s_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 |
| 5 | 0 | 1 | 1 | 1 | 0 |
| 6 | 1 | 0 | 0 | 0 | 1 |
| 7 | 0 | 1 | 0 | 0 | 1 |
| 8 | 0 | 0 | 0 | 1 | 1 |
| 9 | 0 | 1 | 0 | 1 | 1 |
| 10 | x | x | x | x | x |
| 11 | x | x | x | x | x |
| 12 | x | x | x | x | x |
| 13 | x | x | x | x | x |
| 14 | x | x | x | x | x |
| 15 | x | x | x | x | x |

En la figura 20 también puede verse cómo se lleva a cabo la inicialización del circuito, conectando la señal *Inicio* a las entradas asíncronas de los biestables (recordemos que la señal *Inicio* genera un pulso a 1 para indicar al circuito que se ponga en funcionamiento). Dado que en nuestro ejemplo el estado inicial es el *D* (podéis ver la figura 19), al empezar a funcionar el circuito, los biestables toman los valores [$q_2$, $q_1$, $q_0$] = [0 1 1]. El circuito volverá a este estado siempre que *Inicio* haga un pulso.

# Resumen

En este módulo se han estudiado los circuitos lógicos secuenciales. Se ha visto que lo que los caracteriza es su capacidad de memoria y, por tanto, son capaces de determinar el valor de las señales de salida de acuerdo no sólo con el valor actual de las señales de entrada, sino también con el valor que han tenido estas señales de entrada en anteriores momentos.

Se ha visto la necesidad de un mecanismo de sincronización para controlar la evolución temporal de las distintas señales, y se ha presentado la señal de reloj.

Se ha conocido el dispositivo más elemental de memoria, el biestable D, que es capaz de guardar el valor de un bit. Se ha visto que su valor se puede modificar de forma síncrona y asíncrona, gracias a las entradas *R* y *S*. También se ha visto que se puede "congelar" el valor mediante una señal de carga.

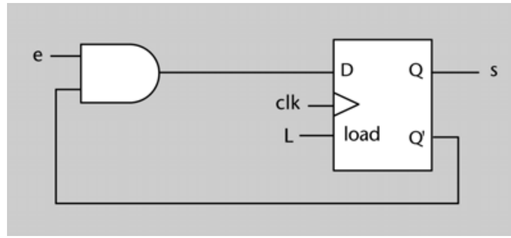
Después se han presentado los diferentes bloques secuenciales que permiten guardar el valor de una palabra (el registro), de un número pequeño de palabras (el banco de registros) o de un gran volumen de palabras (la memoria RAM).

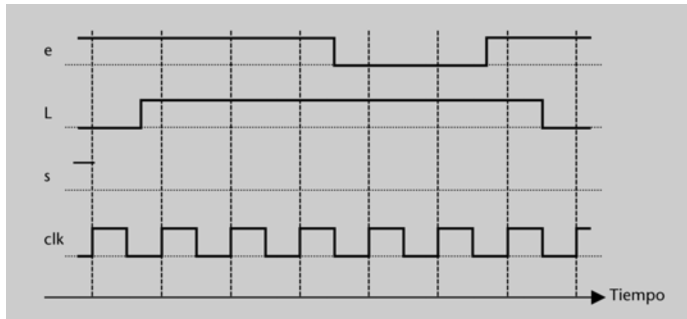
Por último, se ha conocido la forma de especificar el comportamiento de circuitos secuenciales llamada *modelo de Moore*, que se fundamenta sobre los conceptos de *estado* y *transición*. Se ha visto que el comportamiento del circuito se puede expresar mediante las tablas de salidas y transiciones, o bien gráficamente mediante grafos de estados. Se ha aprendido a dibujar la evolución temporal de un circuito sobre un cronograma.

Los bloques secuenciales y combinacionales que se han estudiado en este curso constituyen un conjunto de dispositivos suficiente para diseñar un computador sencillo.

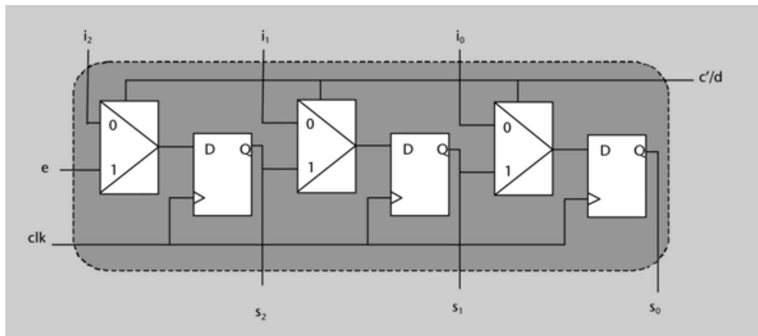
## Ejercicios de autoevaluación

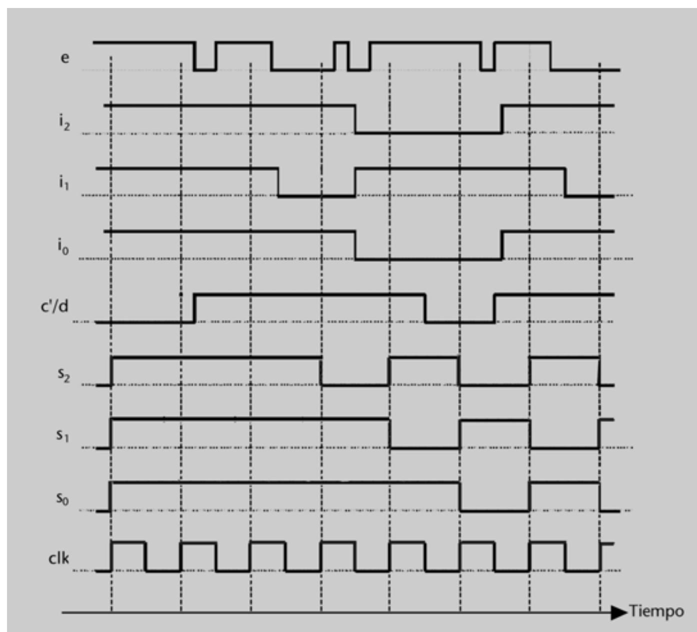
**1.** Completad el cronograma que corresponde al circuito de la figura, suponiendo que inicialmente la salida Q vale 1. ¿Cuál es el papel de la señal *e* en el circuito?



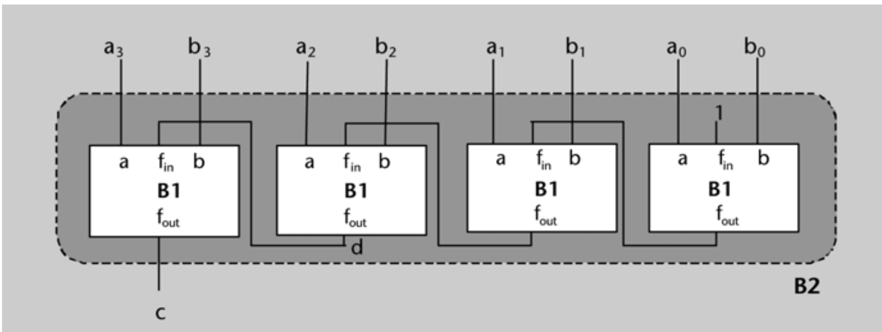

**2.** Completad el cronograma que corresponde al circuito de la figura, suponiendo que inicialmente las salidas Q de todos los biestables valen 0. Describid en pocas palabras qué hace el circuito según la señal *c'/d*.



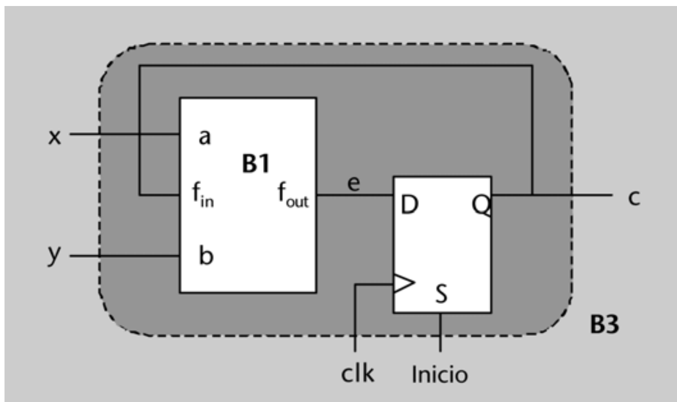



**3.** El circuito combinacional B2 es un comparador de dos números naturales *A* y *B* representados en binario. Su funcionamiento es el siguiente:

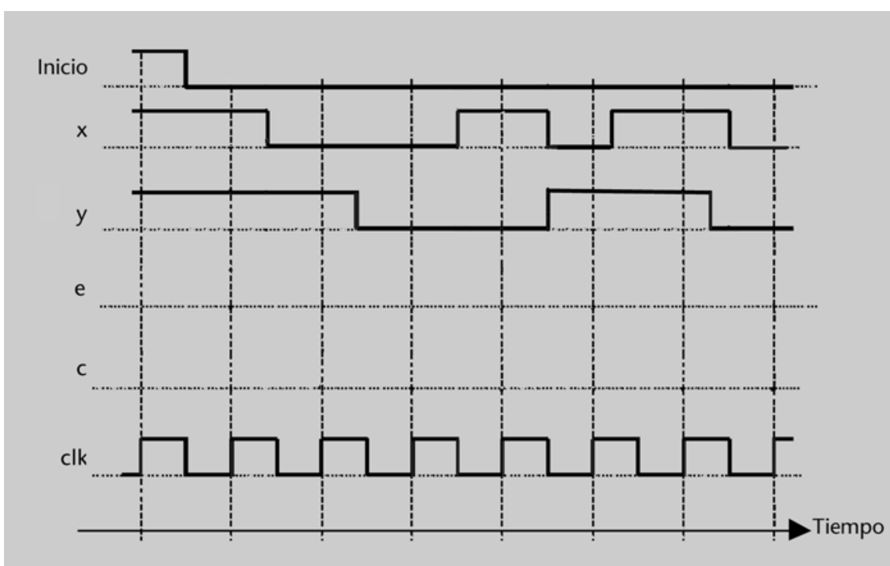
$c$ = 0 si *A* es menor que *B*.
$c$ = 1 si *A* es mayor o igual que *B*.




**a)** Viendo cómo se ha construido el circuito B2 y su funcionalidad, deducid la tabla de verdad del bloque B1.
**b)** Utilizando el mismo bloque B1 se ha construido este otro circuito, B3. Describid qué función hace este circuito, y comparadlo con el circuito B2.



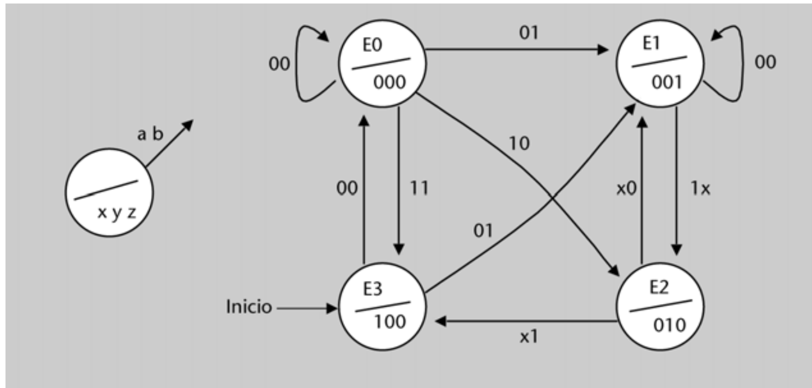

**c)** Completad el siguiente cronograma, que corresponde al circuito B3. Si interpretamos las entradas *x* e *y* en cada ciclo de reloj como los distintos bits de un par de números *A* y *B*, ¿qué números son *A* y *B*?



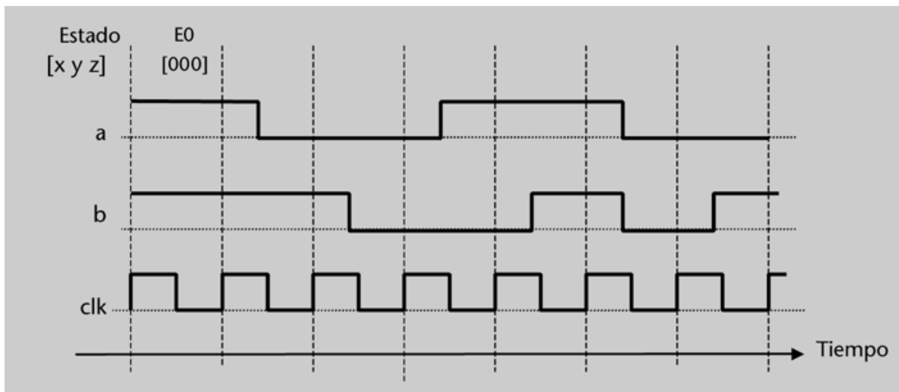

**4.** Se quiere diseñar un circuito secuencial que controle un semáforo. De día (de 8:00 a 20:00), el semáforo tiene que estar en verde durante tres ciclos, en amarillo durante un ciclo y en rojo durante dos ciclos. Por la noche, tiene que estar en verde durante dos ciclos, en amarillo durante un ciclo y en rojo durante tres.

**a)** ¿Qué entradas y salidas debe tener el circuito?
**b)** Expresad el comportamiento del circuito mediante un grafo de estados.

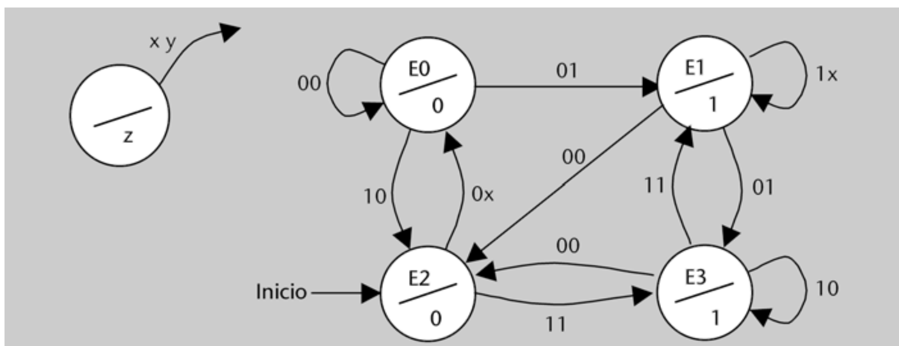
**5.** Dado el siguiente grafo de estados:

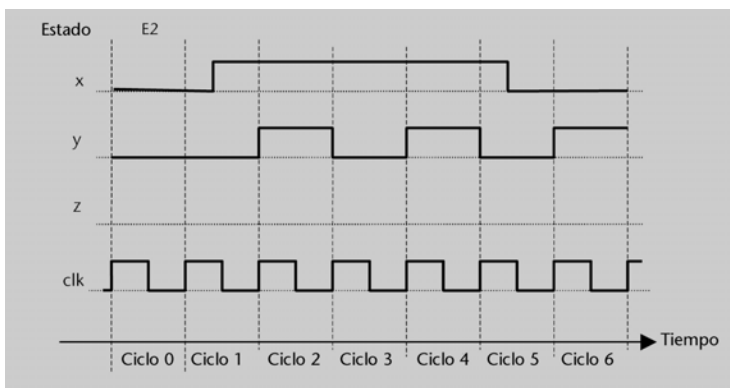



**a)** Escribid la tabla de salidas y la tabla de transiciones del circuito.
**b)** Completad las líneas correspondientes al estado y a las salidas del siguiente cronograma:




**6.** Completad el siguiente cronograma, que corresponde a un circuito con este grafo de estados:



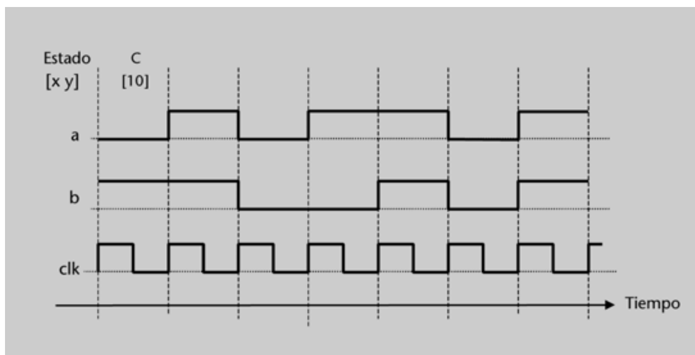



**7.** A continuación se muestran las tablas de salidas y las de transiciones de un circuito secuencial.

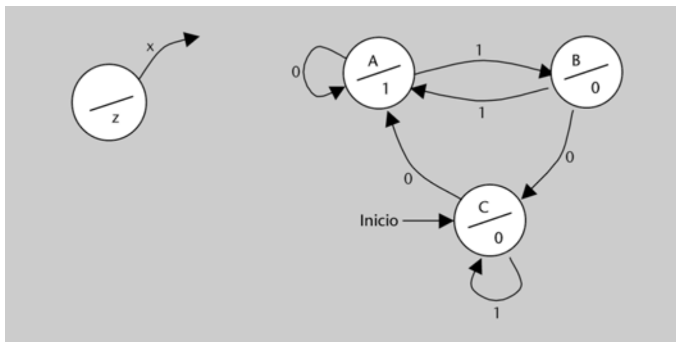
| Tabla de salidas | |
|---|---|
| **Estado** | **xy** |
| A | 0 0 |
| B | 0 1 |
| C | 1 1 |

| Estado | a | b | Estado$^+$ |
|---|---|---|---|
| A | 0 | 0 | B |
| A | 0 | 1 | A |
| A | 1 | 0 | B |
| A | 1 | 1 | A |
| B | 0 | 0 | x |
| B | 0 | 1 | x |
| B | 1 | 0 | C |
| B | 1 | 1 | C |
| C | 0 | 0 | C |
| C | 0 | 1 | B |
| C | 1 | 0 | A |
| C | 1 | 1 | A |

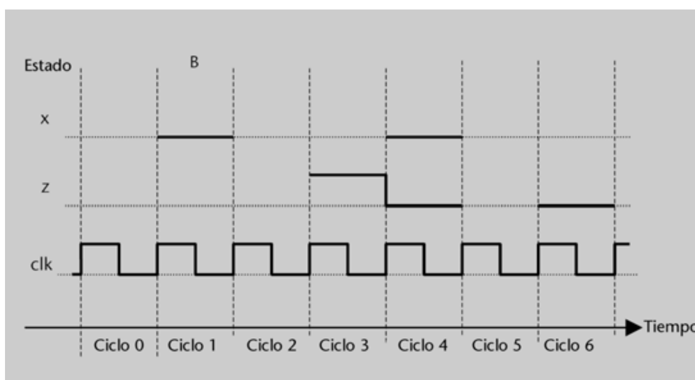
**a)** Dibujad el grafo de estados del circuito, suponiendo que el estado inicial es el *A*.
**b)** Completad las líneas correspondientes al estado y a las salidas del siguiente cronograma:



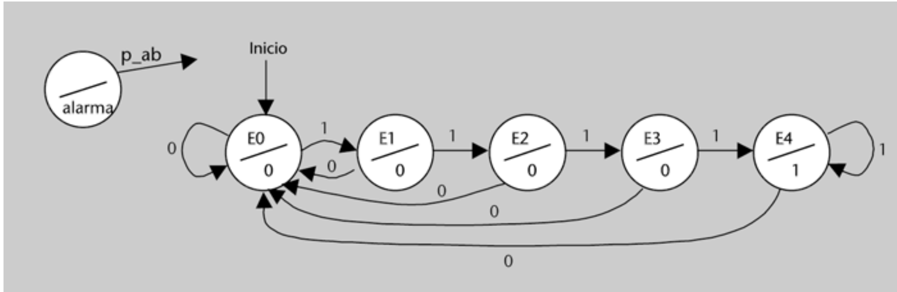

**8.** Sea el siguiente grafo de estados:




Completad el siguiente cronograma. Suponed que la señal de entrada *x* sólo cambia de valor en los instantes de los flancos.



**9.** Un circuito secuencial tiene una entrada *p_ab* y una salida *alarma*, ambas de un bit. La entrada *p_ab* proviene de la caja fuerte de un banco, y vale 1 cuando está abierta y 0 cuando está cerrada. La salida *alarma* está conectada a una señal de alarma, que se activa cuando es *alarma* = 1. El funcionamiento del circuito viene descrito por este grafo de estados:




Si sabemos que el reloj del circuito tiene un periodo de 30 segundos, describid el comportamiento del sistema de alarma de la caja fuerte. Suponed que la puerta de la caja sólo se puede abrir o cerrar cada 30 segundos, coincidiendo con los flancos ascendentes del reloj.

# Solucionario

## Actividades

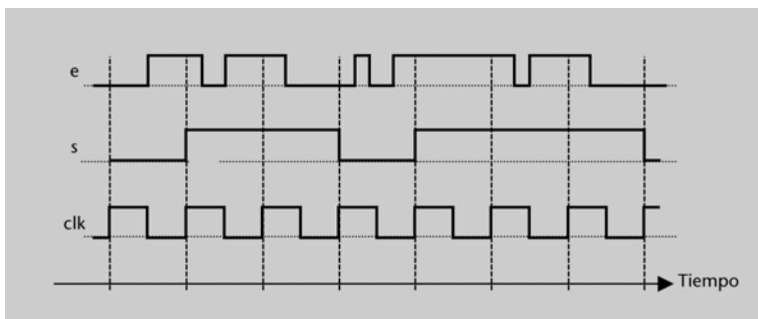
**1.** Se trata de reconocer si los cuatro bits que llegan por la entrada del sistema valen 1010 o no. Para saberlo es suficiente examinar el valor de la palabra de entrada en el momento actual. Por tanto, el sistema es de tipo combinacional.
En cambio, si la entrada de un circuito fuese de un bit, el circuito debería ser de tipo secuencial, porque en cada momento debería recordar los tres últimos valores que han llegado por la entrada, aparte del actual.

**2.** Un dígito decimal (rango 0 al 9) requiere cuatro bits para ser codificado. Por tanto, la entrada del sistema que se tiene que diseñar puede leer un dígito en cada momento.
Dado que se tiene que detectar una secuencia de cuatro dígitos, éstos tienen que entrar uno tras otro por la entrada del sistema, y el circuito tiene que recordar los dígitos que han entrado con anterioridad para reconocer la secuencia.
Se trata, pues, claramente, de un sistema secuencial.

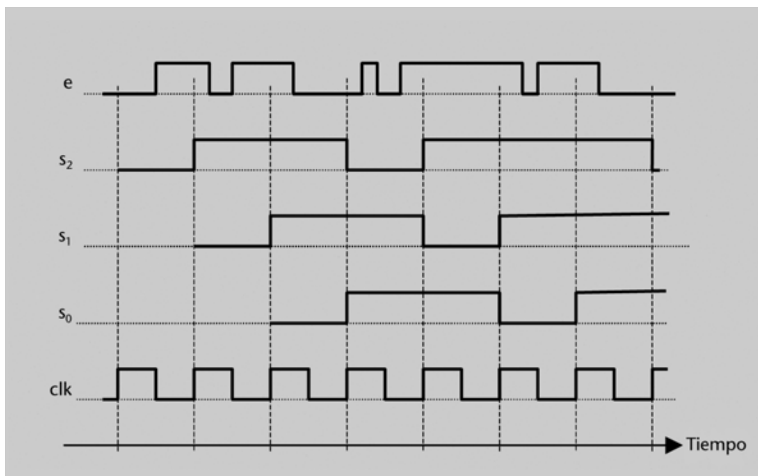
**3.** El periodo $T$ es el inverso de la frecuencia $T = 1/F$.
Puesto que la frecuencia es de $1{,}6 \cdot 10^9$ Hz, el periodo es el siguiente:

$$T = 1/(1{,}6 \cdot 10^9) \text{ s} = 0{,}625 \cdot 10^{-9} \text{ s} = 0{,}625 \text{ ns (nanosegundos)}$$

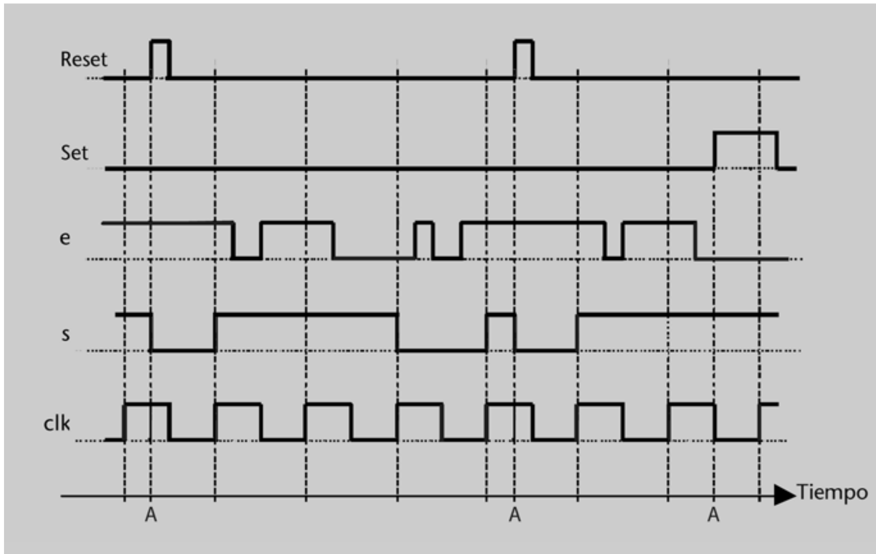
**4.** Como se puede apreciar en el cronograma, en la salida $s$ del biestable está el valor leído de la entrada $e$ en cada flanco ascendente del reloj.
Fijémonos en que la salida del biestable sólo cambia en los flancos ascendentes del reloj (es decir, cambia de forma síncrona), mientras que la entrada puede cambiar en cualquier momento (es decir, cambia de forma asíncrona).
Es importante notar que el valor de $s$ no se conoce antes del primer flanco de reloj, porque no se conoce el valor que había en la entrada $D$ del biestable en el instante del flanco anterior. Por esta razón, $s$ no tiene valor.



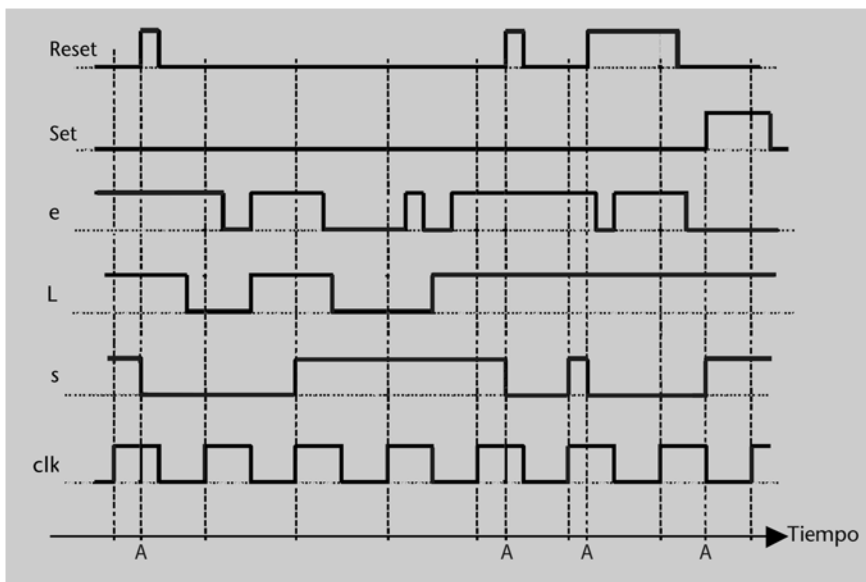

**5.** En este caso se puede ver que la entrada $e$ se "desplaza" de forma sucesiva por las salidas $s_2$, $s_1$ y $s_0$ de los biestables.
La mejor forma de hacer este tipo de cronogramas es dibujar primero la línea correspondiente a la señal $s_2$, que sólo depende de $e$, entera. Una vez ésta se ha dibujado, se dibuja la señal $s_1$, que depende sólo de la señal $s_2$. Por último, se dibuja la línea correspondiente a la señal $s_0$, que depende sólo de la señal $s_1$.
Notamos que el valor de $s_1$ no se puede determinar hasta el segundo flanco del reloj, y el de $s_0$ hasta el tercero.



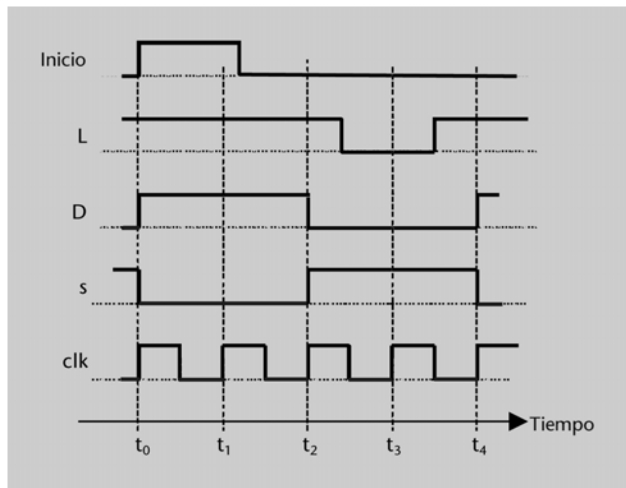
**6.** Para resolver este ejercicio debemos tener en cuenta que las entradas asíncronas tienen prioridad sobre las síncronas, es decir, primero se evalúan los valores de las entradas $R$ y $S$ de los biestables, y sólo cuando ambas están a 0 se evalúa el valor presente en la entrada $D$ en cada flanco de reloj (dado que el biestable no tiene señal de carga, asumimos que está a 1).
Por esta razón, en el cronograma aparecen marcados con líneas verticales con la letra A los momentos en que cambian las entradas asíncronas, además de los flancos ascendentes del reloj.




**7.** A la entrada $D$ del biestable llega o bien la salida del biestable (cuando $L$ vale 0), o bien la señal $e$ (cuando $L$ vale 1).
Dado que el biestable se carga en cada flanco de reloj, tenemos que cuando $L = 0$ se carga lo mismo que había. El efecto es que su salida $Q$ no cambia.
Por el contrario, cuando $L = 1$, se almacena en el biestable el valor de la señal $e$.
Por tanto, podemos decir que este circuito se comporta igual que un biestable con señal de carga, en el que la señal $L$ hace el papel de la entrada *load*.
Fijémonos en que después del último flanco el biestable se mantiene a 1, aunque $L = 1$ y $e = 0$. Esto se debe al hecho de que *Set* está a 1.
Como en la actividad 6, en el cronograma aparecen marcados los momentos en que cambian las entradas asíncronas con la letra A además de los flancos de subida del reloj.

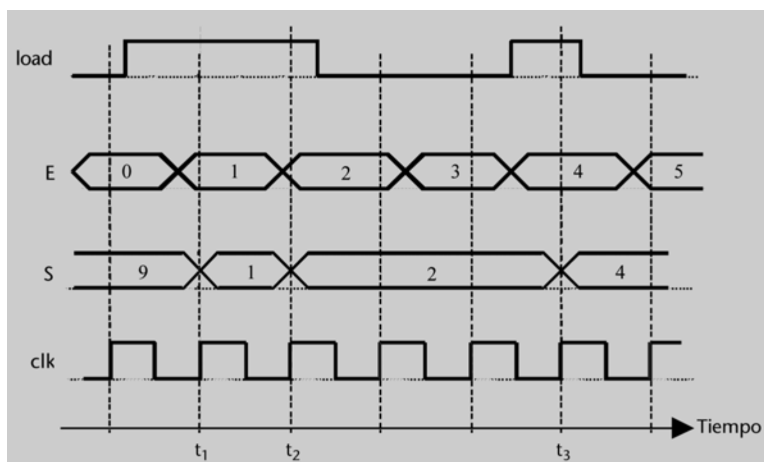



**8.** Inicialmente, $D = 0$, porque $s = 1$. En el instante $t_0$, *Inicio* se pone a 1, por tanto, el biestable se pone a 0. Se quedará al menos hasta $t_2$, porque *Inicio* continúa valiendo 1. En $t_2$ se carga con lo que hay en la entrada $D$, es decir, un 1. En $t_3$, el biestable no se carga, porque $L = 0$. En $t_4$ se carga con lo que hay en la entrada $D$, es decir, un 0.




**9.** Podemos observar en el cronograma que el registro sólo se carga en los flancos en los que *load* = 1: $t_1$, $t_2$ y $t_3$. En estos instantes, se carga con el valor de la entrada *E*. La secuencia de valores que toma la salida del registro, tal como se muestra en el cronograma, es la siguiente:

9 1 2 4.

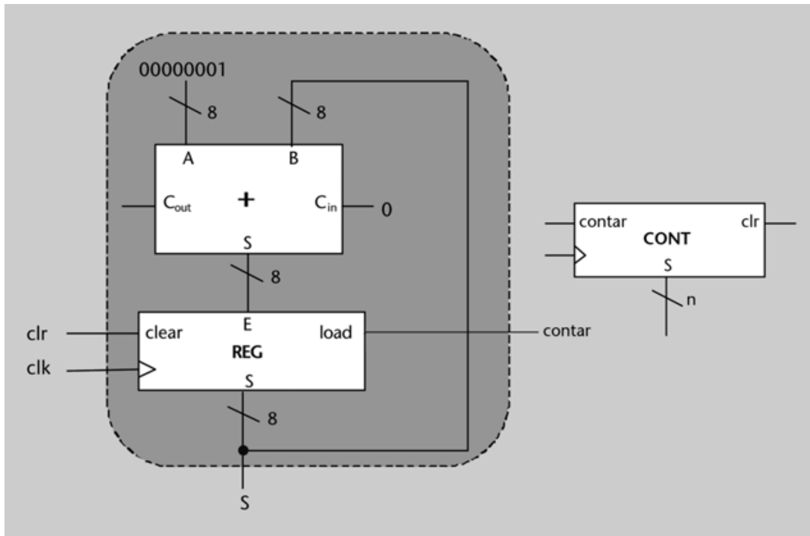



**10.**

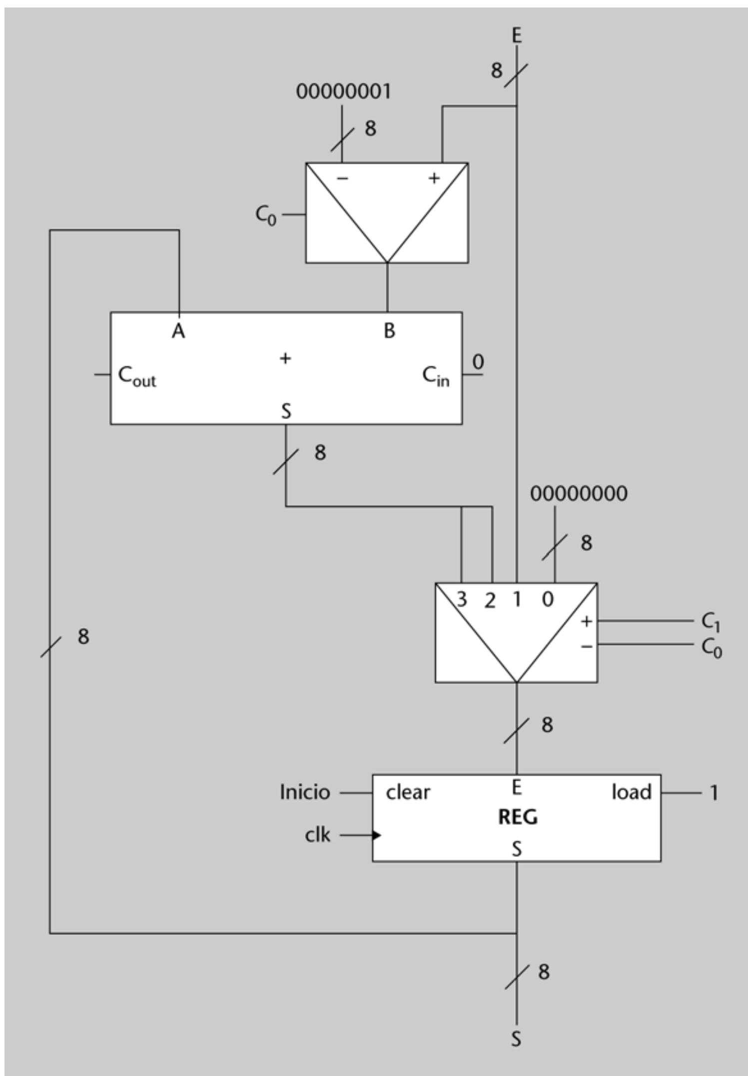
**a)** En este circuito, la salida del sumador se conecta directamente a una de sus entradas. Esto implica que *S* nunca será estable, continuamente estará variando.

**b)** En este caso, *S* expresa correctamente la suma acumulada de los valores que ha tomado *X* (en los instantes anteriores a cada flanco, tal y como indica la convención que se ha establecido en estos apuntes) desde la inicialización del circuito. Sin embargo, en el momento en el que esta suma acumulada sea mayor que 255 el valor de *S* dejará de ser correcto, porque no basta con 8 bits para representarlo. Podríamos pensar en hacer que el registro sea de más de 8 bits, pero sea cual sea el número de bits que tenga, siempre llegará un momento en el que se producirá desbordamiento (a no ser que *Inicio* haga un nuevo pulso a 1 antes de producirse el desbordamiento, pero que esto pase o no es incontrolable).

**11.** Usaremos un registro que contendrá en todo momento la salida *S* del circuito. Siempre que *contar* valga 1, su contenido se debe incrementar en una unidad en cada flanco del reloj, por lo cual conectaremos la salida del registro a la entrada de un sumador. A la otra entrada del sumador, conectaremos un 1. La salida del sumador está conectada a la entrada del registro para que se guarde en el próximo flanco de reloj si *contar* vale 1. Por lo tanto, conectaremos *contar* a la entrada *load* del registro.
Otra posibilidad es poner un 0 a la otra entrada del sumador y sumar el 1 por medio de la entrada de transporte del sumador, que en el diseño mostrado tiene un 0.
Por otro lado, la señal *clr* estará conectada a la entrada *clear* del registro para ponerlo a 0 de manera asíncrona.
La figura muestra el circuito que conforma el bloque, y el dibujo del bloque con sus entradas y salidas.




**12.** El valor de *S* se debe actualizar en cada ciclo, y por lo tanto tiene que estar guardado en un registro, a cuya entrada *clear* conectamos la señal *Inicio* y a cuya entrada *load* conectamos un 1. Para determinar qué valor se cargará a cada flanco ascendente, usaremos un multiplexor 4-1 gobernado por las señales de control $c_1$ y $c_0$. Cuando $c_1 = 1$, el valor que se debe cargar al registro es su contenido actual sumado con 1 (si $c_0 = 0$) o *E* (si $c_0 = 1$). Esto lo conseguiremos con un sumador, a una de cuyas entradas conectamos *S* y a la otra conectamos la salida de un multiplexor 2-1 gobernado por $c_0$.



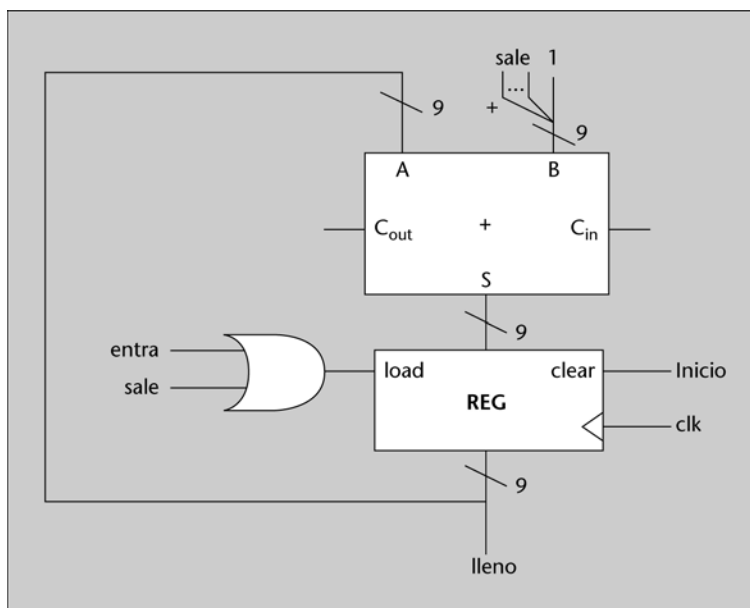
**13.**

**a)** El circuito debe saber cuándo entra o sale un coche, y por lo tanto *entra* y *sale* deben ser señales de entrada, así como también la señal *Inicio*. Y su misión es dar en todo momento el valor correcto a la señal *lleno*, que será su señal de salida. Todas las señales son de un bit.

**b)** El circuito debe ser secuencial porque debe saber en todo momento cuántos coches hay en el aparcamiento, para poder determinar si ya está lleno o no. Por lo tanto, debe tener memoria de cuántos han entrado y salido desde la inicialización y hasta el momento actual.

**c)** El circuito debe tener un registro que guarde el número de vehículos que hay en el aparcamiento. Puesto que puede haber 500 como mucho, el registro debe tener 9 bits. Conectaremos *Inicio* a su entrada *clear*.

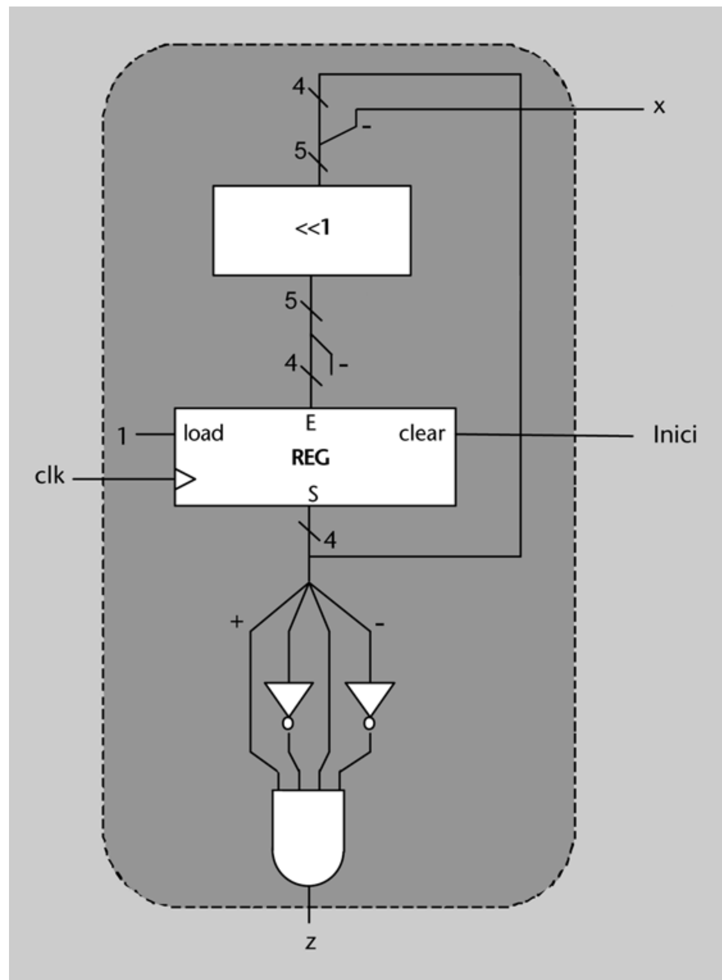
Este registro se debe incrementar o decrementar en una unidad siempre que se produzca un pulso en la señal *entra* o *sale*, respectivamente. Por lo tanto, la entrada *load* debe estar a 1 cuando alguna de estas dos señales esté a 1. Habrá que disponer de un sumador que sume el contenido del registro más 1 o –1 dependiendo del caso. La suma en binario y en complemento a 2 se hacen de la misma manera (es decir, el resultado de una suma será correcto tanto si interpretamos las entradas y salida del sumador en binario como en complemento a 2). Por lo tanto, en caso de que queramos sumar 1 podemos poner a la entrada del sumador 000000001, y en caso de que queramos sumar –1 podemos poner 111111111 (que es –1 codificado en complemento a 2). Teniendo en cuenta que siempre que *sale* valga 1 *entra* valdrá 0, podemos conseguir el valor deseado en la entrada *B* del sumador tal y como se muestra en el circuito: el bit de menos peso vale siempre 1 (recordemos que el registro sólo se cargará si *entra* o *sale* valen 1), y los otros 8 bits se forman replicando 8 veces la señal *sale*.




**14.** Queremos reconocer si en la entrada *x* de un bit se produce la secuencia de valores 1010, y disponemos de un registro de cuatro bits. Por tanto, tenemos que hacer que los valores que lleguen por la entrada *x* se desplacen por los biestables del registro, por ejemplo, de derecha a izquierda. El biestable que contiene el bit de menos peso se cargará en cada flanco con el valor que contenía su "vecino" de la derecha.
Para conseguirlo, haremos que el registro se cargue en cada flanco con el valor que contenía hasta ahora desplazado un bit a la izquierda, excepto por el bit situado a la derecha de todos, que se cargará con el nuevo valor de *x*.
De esta forma obtenemos que la salida *z* debe valer 1 cuando el contenido del registro sea 1010. Si llamamos $[s_3\ s_2\ s_1\ s_0]$ a los bits de salida del registro, tenemos lo siguiente:

$$z = s_3 \cdot s_2' \cdot s_1 \cdot s_0'.$$

Inicialmente, el registro tiene que contener un 0, ya que de otra forma se podría reconocer la secuencia sin que se hubiese producido. Por ejemplo, si el contenido inicial del registro fuese 0101 y el primer valor de *x* fuese 0, el circuito reconocería la secuencia 1010 después del primer flanco de reloj, de forma errónea. Por eso conectamos la señal *Inicio* en la entrada *clear* del registro.




**15.** Analizaremos punto por punto el circuito:

- La entrada *X* se carga en el registro *REG1* en cada flanco de reloj (dado que no se han dibujado las entradas *load* y *clear*, se asume que valen 1 y 0, respectivamente).
- El bit de más peso del registro *REG1*, $R_{n-1}$, controla el multiplexor.
- Dado que los números están representados en complemento a 2, el bit $R_{n-1}$ es el bit de signo del último número que se ha almacenado en el registro *REG1*. Este bit es 1 si el número es negativo y 0 si es positivo.
  Por tanto, el multiplexor deja pasar el número que hay en *REG1* si éste es positivo, y deja pasar un 0 si es negativo.
- El sumador suma este número con el contenido de registro de *REG2*, que inicialmente está a 0, y el resultado se guarda otra vez en *REG2*.
- La salida *S* del circuito está conectada a la salida de *REG2*.

Podemos concluir, pues, que la salida *S* del circuito es la suma de todos los números positivos que entran por *X*.

**16.** Vemos que la señal *Inicio* está conectada a todas las entradas *clear* de los registros. Por tanto, todos se ponen a 0 cuando empieza a funcionar el circuito.
Las entradas *load* de cada registro están conectadas a las salidas de un descodificador. Por tanto, sólo una de éstas está a 1 en cada momento, si $e = 1$, (la señal $e$ está conectada a la entrada de validación del descodificador). Si $e = 0$, entonces no se carga ningún registro.
En las entradas del descodificador están las señales $c_1$ y $c_0$. Por tanto, deducimos que estas dos señales controlan qué registro se carga en cada momento con el valor de la entrada *X*, que está conectada a la entrada de datos de todos los registros.
La salida *S* del circuito está conectada a un multiplexor de buses controlado por $c_3$ y $c_2$. Cada entrada de datos del multiplexor está conectada a la salida de datos de uno de los cuatro registros. Por tanto, deducimos que $c_3$ y $c_2$ controlan qué contenidos de los cuatro registros sale en cada momento por la salida.

**17.**

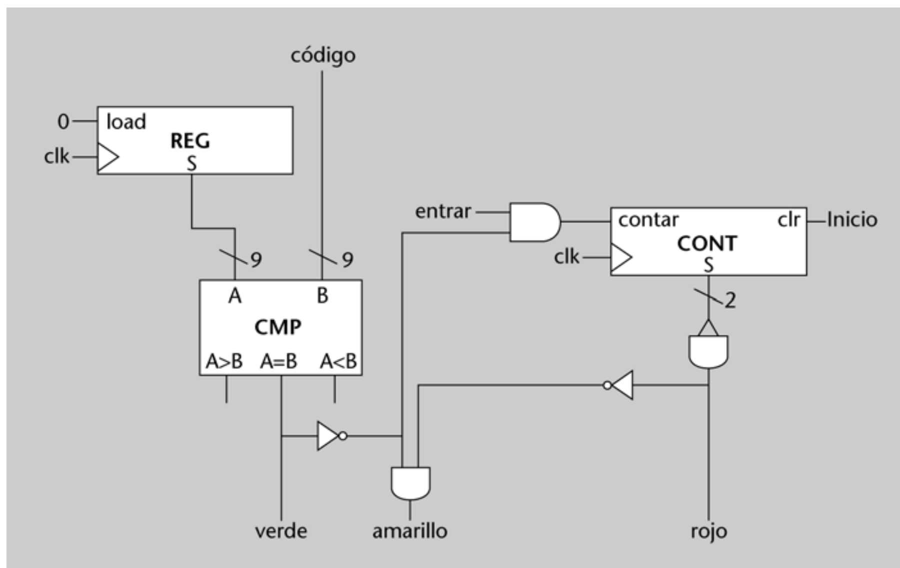
**a)** Las entradas son:

- *entrar*, de un bit.
- *código*, de 9 bits, porque el máximo valor que puede tener es 444.
- *Inicio*, de un bit.

Y las salidas son *verde*, *amarillo* y *rojo*, las tres de un bit.

**b)** El circuito debe tener un comparador para comparar el código que se ha tecleado con la contraseña correcta. Cuando la salida A=B valga 1, se debe activar la señal *verde*, mientras que si vale 0 se deben activar o bien *amarillo* o bien *rojo*, dependiendo de cuántas veces se haya introducido un código incorrecto.

Para saber cuántas veces se ha tecleado un código incorrecto, usaremos un bloque contador como el que se diseña en la actividad 11; puesto que sólo debe contar hasta 3, puede tener sólo 2 bits. Se tendrá que incrementar en 1 siempre que la salida A=B del comparador valga 0 y se haya introducido un nuevo código; esto lo conseguimos conectando a la entrada *contar* del contador la salida de una puerta AND, a cuyas entradas conectamos *entrar* y la negación de la salida A=B del comparador. Observemos que, puesto que *entrar* vale 1 sólo durante un ciclo de reloj, el contador se incrementará sólo una vez para cada nuevo código que se teclee. La señal *rojo* se deberá activar cuando la salida del contador sea 11, y cuando esto pase la señal *amarillo* debe estar a 0. La luz amarilla se debe encender cuando el código tecleado no coincida con la contraseña correcta pero el contador aún no haya llegado a 3.
La figura muestra el circuito completo. El registro de la izquierda es el que contiene la contraseña correcta.

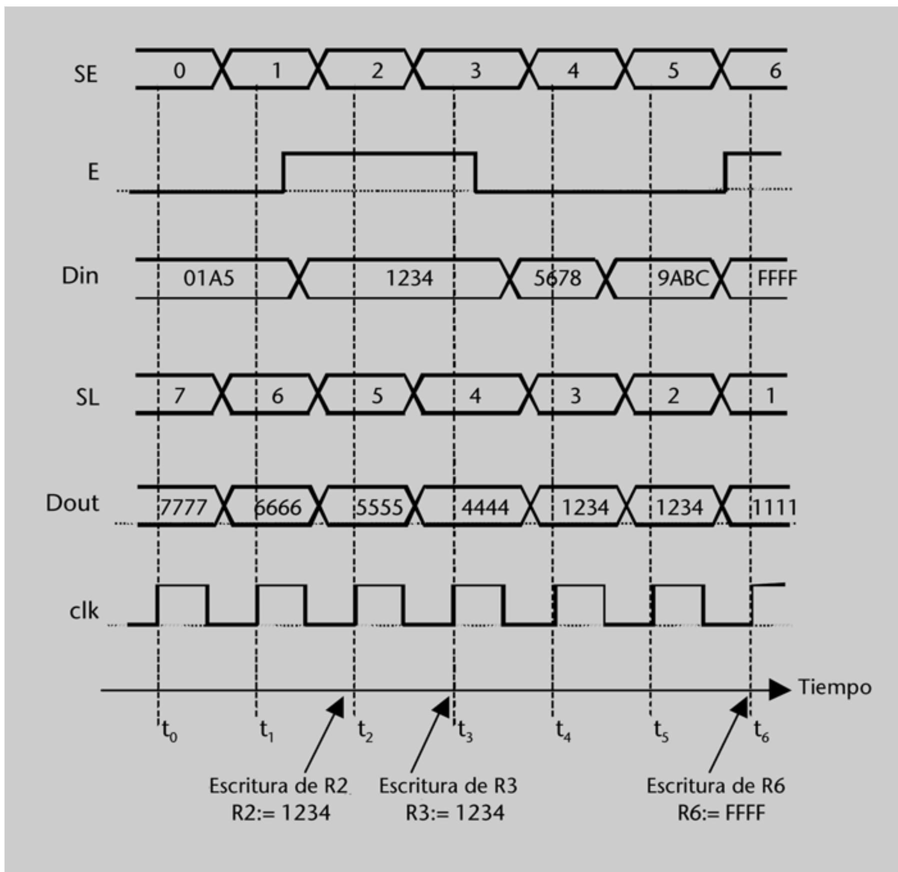



**c)** Sería necesario definir una señal *cambio_contraseña* que se activaría cuando se quisiera hacer el cambio y que se debería conectar a la entrada *load* del registro que guarda la contraseña correcta. También sería preciso establecer alguna manera de teclear la nueva contraseña, y hacer llegar su valor codificado en binario a la entrada de datos del registro.

**18.**
**a)** Las escrituras en el banco de registros tienen lugar en los flancos ascendentes si $E = 1$. Por tanto, los instantes en que se escribirá algún registro son $t_2$ $t_3$ y $t_6$. Para saber qué registro se escribe analizamos el valor de *SE* en estos instantes; para saber qué valor se carga, analizaremos *Din*. Obtenemos lo siguiente:

En el instante $t_2$, `R2 := 1234`
En el instante $t_3$, `R3 := 1234`
En el instante $t_6$, `R6 := FFFF`

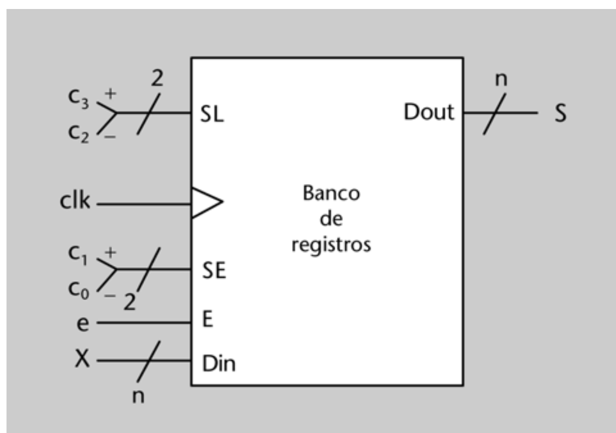
**b)** En el cronograma se puede ver la secuencia de valores en *Dout*. Para deducirlo, hay que observar en cada momento qué registro se lee (*SL*) y cuál es su contenido (recordemos que las lecturas se hacen de forma asíncrona).




**c)** El valor final de los registros es el siguiente:

*R0* = 0000,
*R1* = 1111,
*R2* = 1234,
*R3* = 1234,
*R4* = 4444;
*R5* = 5555;
*R6* = FFFF;
*R7* = 7777.

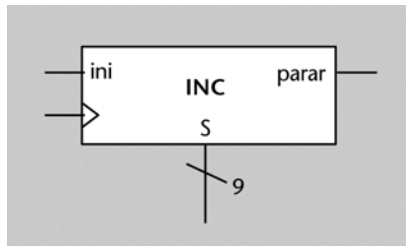
**19.** La funcionalidad del circuito de la actividad 16 (sin la entrada *Inicio*) se consigue con un banco de registros que conecta las señales a las entradas y salidas tal como se muestra en la figura.



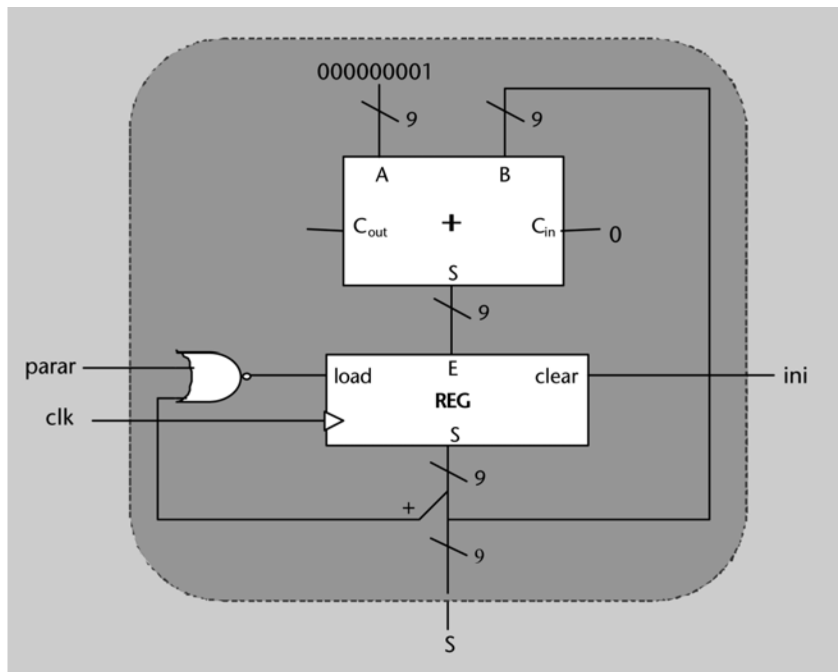

Deducimos, pues, que el circuito de la actividad 16 corresponde a una posible implementación de un banco de cuatro registros, al que se le añade la funcionalidad de inicializar todos los registros a 0.

**20.**

**a)** El circuito que se tiene que diseñar es el siguiente:

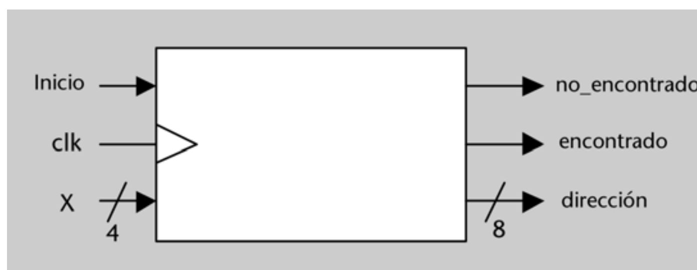



– Para hacer este bloque utilizaremos, básicamente, un registro y un sumador. El registro almacenará la salida *S* y el sumador permitirá incrementarla. Para hacer esto, a una entrada del sumador llegará el valor de *S* y a la otra, un 1.
– Implementaremos la señal *ini* con el *clear* del registro.
– Implementaremos la señal *parar* con la señal de *load* del registro. Cuando *parar* sea 1 o el bit de más peso de *S* sea 1, pondremos un 0 en *load*. En cualquier caso, la señal de *load* será 1. Por tanto, *load* = (*parar* + $S_8$)'.




**b)** El circuito tiene tres entradas y tres salidas.
Entradas:
– Una entrada (*Inicio*) de un bit, que inicializa el circuito.
– Una entrada de reloj: *clk*.
– Una entrada de datos de cuatro bits: *X*.
Salidas:
– Una salida de ocho bits: *dirección*.
– Dos salidas de un bit: *encontrado, no_econtrado*.

A continuación se muestra una figura con esta descripción.




**c)** El bloque CMP es combinacional, y el resto son secuenciales.

**d)** El tamaño de la RAM es de $2^8 \cdot 4$ bits.
**e)** Para ver qué hace el circuito, lo analizaremos por partes:
– Cuando el circuito empieza a funcionar, la entrada *X* se almacena en el registro (porque *Inicio* está conectado en la entrada *load*).
– El bloque INC se incrementa en cada flanco de reloj. Empieza desde 0 gracias a la señal *Inicio*.
– La salida del bloque INC está conectada a la entrada de direcciones de la memoria. Puesto que *L'/E* = 0, en cada ciclo de reloj se lee una palabra de la memoria, desde la dirección 0 hasta la 255. Si el bloque INC llega a 256 (100000000), parará de incrementarse, y se habrá recorrido toda la memoria. En este momento, el bit de más peso del bloque INC (no-encontrado) valdrá 1, y en *M@* habrá un 0 (los ocho bits de menor peso de INC).
– El bloque INC también se puede parar antes si la entrada *parar* se pone a 1. Esto pasa cuando la señal *encontrado* vale 1.
– La salida de datos de la RAM se compara en cada instante con el contenido del registro (la entrada *X*). Si son iguales, la señal *encontrado* se pone a 1.
– Dado que *encontrado* detiene el incrementador, éste se para cuando se ha encontrado una palabra en la memoria que es igual a la palabra de entrada *X*. En este momento, la salida *dirección* contiene la posición de memoria donde se encuentra la palabra *X*.

Podemos concluir que lo que hace este circuito es buscar en qué posición de memoria se encuentra una palabra *X*. Si esta palabra está en la memoria, la señal *encontrado* se pone a 1 y por la salida *dirección* sale la posición de memoria donde se ha encontrado. Si la palabra no está en la memoria, la señal *no_encontrado* se pone a 1 y el contenido de la salida *dirección* no tiene ningún significado.

**21.**
**a)** El coche puede estar en cuatro situaciones: parado, girando hacia la derecha, girando hacia la izquierda y moviéndose adelante. El circuito tendrá, pues, cuatro estados, que llamaremos respectivamente *PARADO, DERECHA, IZQUIERDA* y *ADELANTE*.
**b)** Lo más habitual es que, al pulsar el botón "on" del coche para empezar a jugar, éste permanezca parado. Por tanto, podemos decir que el estado inicial es *PARADO*.
**c)** A partir de la tabla que describe la acción de las señales $z_1$ y $z_0$ sobre el coche, obtenemos que las señales de salida deben tener los siguientes valores:

| **Estado** | $z_1$ | $z_0$ |
|---|---|---|
| PARADO | 1 | 0 |
| DERECHA | 0 | 0 |
| IZQUIERDA | 0 | 1 |
| ADELANTE | 1 | 1 |

El enunciado nos dice que las transacciones que se producirán serán las siguientes:

| **Estado** | **e** | **d** | **Estado$^+$** |
|---|---|---|---|
| PARADO | 0 | 0 | PARADO |
| PARADO | 0 | 1 | DERECHA |
| PARADO | 1 | 0 | IZQUIERDA |
| PARADO | 1 | 1 | ADELANTE |
| DERECHA | 0 | 0 | DERECHA |
| DERECHA | 0 | 1 | DERECHA |
| DERECHA | 1 | 0 | ADELANTE |
| DERECHA | 1 | 1 | PARADO |
| IZQUIERDA | 0 | 0 | IZQUIERDA |
| IZQUIERDA | 0 | 1 | ADELANTE |
| IZQUIERDA | 1 | 0 | IZQUIERDA |
| IZQUIERDA | 1 | 1 | PARADO |
| ADELANTE | 0 | 0 | ADELANTE |
| ADELANTE | 0 | 1 | DERECHA |
| ADELANTE | 1 | 0 | IZQUIERDA |
| ADELANTE | 1 | 1 | PARADO |

**22.**
**a)** El circuito necesita saber si los valores de *x* e *y* han sido iguales en al menos tres ocasiones.
• Para esto tiene que ser capaz de recordar estas situaciones:
• Los valores de *x* e *y* no han sido iguales en ninguna ocasión.
• Lo han sido en una ocasión.

- Lo han sido en dos ocasiones.
- Lo han sido en tres ocasiones o más.

Éstos serán los estados del circuito, que llamaremos respectivamente, *NINGUNA, UNA, DOS* y *TRES_O_MÁS*.

**b)** Cuando el circuito se ponga en funcionamiento, puesto que aún no habrá llegado a ningún valor por las entradas, tendrá que encontrarse en el estado *NINGUNA*.

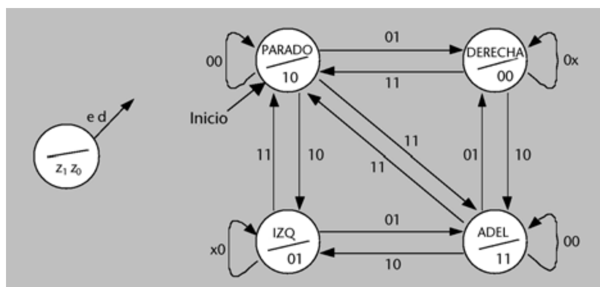
**c)** La salida sólo tiene que valer 1 cuando $x$ e $y$ hayan sido iguales al menos tres veces y, por tanto, la tabla de salidas es la siguiente:

| Estado | z |
|---|---|
| NINGUNA | 0 |
| UNA | 0 |
| DOS | 0 |
| TRES_O_MÁS | 1 |

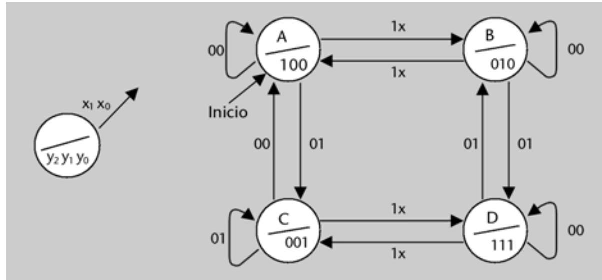
Siempre que $x$ e $y$ sean diferentes, el circuito permanecerá en el estado en que se encuentre. Cuando sean iguales, pasará a recordar que han sido iguales en una ocasión más. Una vez haya llegado al estado *TRES_O_MÁS*, ya no saldrá de allí. Por tanto, la tabla de transiciones es ésta:

| Estado | x | y | Estado$^+$ |
|---|---|---|---|
| NINGUNA | 0 | 0 | UNA |
| NINGUNA | 0 | 1 | NINGUNA |
| NINGUNA | 1 | 0 | NINGUNA |
| NINGUNA | 1 | 1 | UNA |
| UNA | 0 | 0 | DOS |
| UNA | 0 | 1 | UNA |
| UNA | 1 | 0 | UNA |
| UNA | 1 | 1 | DOS |
| DOS | 0 | 0 | TRES_O_MÁS |
| DOS | 0 | 1 | DOS |
| DOS | 1 | 0 | DOS |
| DOS | 1 | 1 | TRES_O_MÁS |
| TRES_O_MÁS | X | X | TRES_O_MÁS |

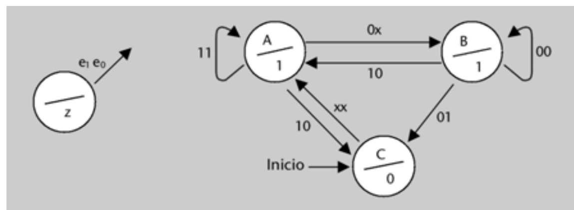
**23.**
Llamaremos “IZQ”, al estado *IZQUIERDA* y “ADEL”, al estado *ADELANTE*. Recordemos que el estado inicial es *PARADO*; entonces traducimos directamente de las tablas de transiciones y salidas que hemos obtenido en la actividad 21 y podemos dibujar el siguiente grafo:



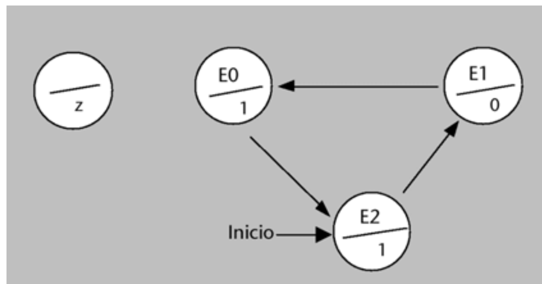
**24.**
Si traducimos directamente de las tablas de transiciones y de salidas, podemos dibujar los siguientes grafos:
**a)**




**b)** En este caso, estando los estados en *B* o *C*, no se producirá nunca la combinación de entrada [1 1]. Por otro lado, del estado *C* pasamos siempre al estado *A*.




**c)** Fijémonos en que este circuito no tiene ninguna señal de entrada.




**25.**
**a)** Vemos que se trata de un circuito sin ninguna señal de entrada. Fijémonos en que hemos escrito en la tabla de salidas las señales en un orden diferente del que aparece en la leyenda del grafo (los podemos escribir en cualquier orden, siempre que mantengamos la coherencia).

| Tabla de transiciones | |
|---|---|
| **Estado** | **Estado$^+$** |
| A | B |
| B | C |
| C | A |

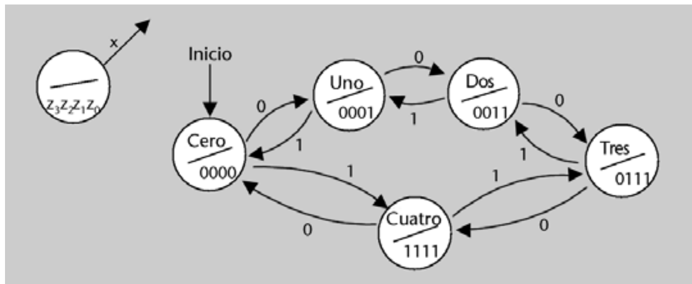
| Tabla de salidas | | | |
|---|---|---|---|
| **Estado** | **x** | **y** | **z** |
| A | 1 | 0 | 0 |
| B | 0 | 1 | 0 |
| C | 0 | 0 | 1 |

**b)** Si observamos el grafo, vemos que estando en el estado *B*, la entrada no vale nunca 1. Pese a todo, incluimos en la tabla de transiciones la fila correspondiente a esta combinación.

| Tabla de transiciones | | |
|---|---|---|
| **Estado** | **a** | **Estado$^+$** |
| A | 0 | B |
| A | 1 | A |
| B | 0 | C |
| B | 1 | x |
| C | 0 | A |
| C | 1 | A |

| Tabla de salidas | | |
|---|---|---|
| **Estado** | **p** | **q** |
| A | 0 | 0 |
| B | 1 | 0 |
| C | 0 | 1 |

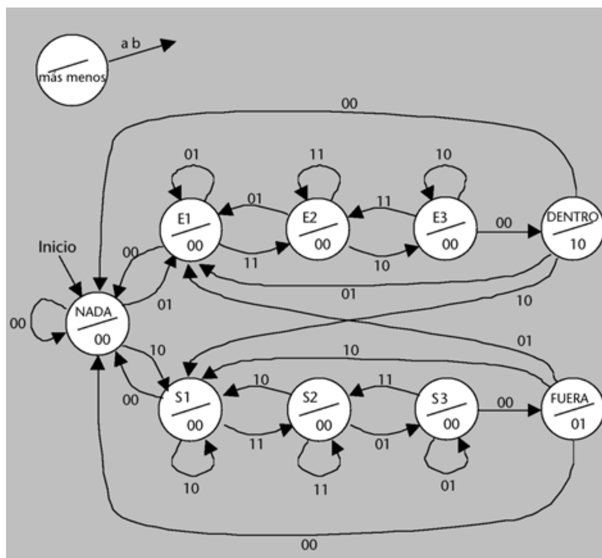
**26.**
El grafo de estados es el siguiente:




**27.**
Para entrar en el aparcamiento, un coche tiene que pasar cuatro etapas: primero pasará por delante del sensor *B*, después estará ante ambos sensores, después, sólo ante el sensor *A* y, por último, entrará en el aparcamiento. Para salir también tendrá que pasar por cuatro etapas análogas, pero en sentido contrario. En consecuencia, para poder determinar el valor de las señales *más* o *menos*, el circuito tiene que ser capaz de distinguir las situaciones que se muestran en la tabla siguiente. Se incluye también la tabla de salidas del circuito; vemos que la señal *más* sólo se tiene que poner a 1 cuando un coche ya ha entrado (estado *DENTRO*), y la señal *menos* sólo se tiene que poner a 1 cuando un coche ya ha salido (estado *FUERA*). El estado inicial es *NADA*.

| Estado | Nombre | más | menos |
|---|---|---|---|
| No hay ningún movimiento | NADA | 0 | 0 |
| Entra un coche, etapa 1 | E1 | 0 | 0 |
| Entra un coche, etapa 2 | E2 | 0 | 0 |
| Entra un coche, etapa 3 | E3 | 0 | 0 |
| Ha entrado un coche completamente | DENTRO | 1 | 0 |
| Sale un coche, etapa 1 | S1 | 0 | 0 |
| Sale un coche, etapa 2 | S2 | 0 | 0 |
| Sale un coche, etapa 3 | S3 | 0 | 0 |
| Ha salido un coche completamente | FUERA | 0 | 1 |

Las transiciones entre estados se muestran en el siguiente grafo. Vemos que hay estados en los que no se darán algunas combinaciones de entrada. Por ejemplo, en el estado E1 (hay un coche que está ante el sensor *B*) no se dará nunca la combinación [*a b*] = [1 0] (porque el enunciado dice que nunca se encontrarán dos coches de cara).
Recordemos que un coche puede parar o hacer marcha atrás en cualquier momento. Por eso, por ejemplo, si en el estado *E2* (el coche está ante dos sensores) se produce la combinación [*a b*] = [0 1], vamos al estado *E1* (el coche ha hecho marcha atrás hasta situarse ante el sensor *B*). Si se produce la combinación [*a b*] = [1 1], nos quedaremos en el estado *E2* (el coche continúa ante los dos sensores).

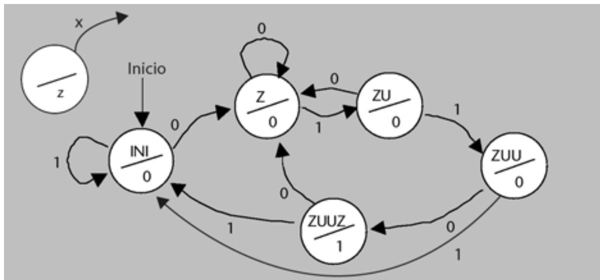


**28.**

**a)** El grafo de estados se muestra a continuación. Utilizamos los siguientes mnemotécnicos para los nombres de los estados:

| Estado | nombre |
|---|---|
| Estado inicial | INI |
| Ha llegado un 0 | Z |
| Ha llegado la subsecuencia 01 | ZU |
| Ha llegado la subsecuencia 011 | ZUU |
| Ha llegado la secuencia 0110 | ZUUZ |

Fijémonos en que en el estado *ZUUZ* se pasa siempre a un estado en que la salida vale 0 y, por tanto, cuando se reconoce la secuencia, la salida está a 1 durante un único ciclo de reloj.




**29.**

Este grafo de estados es muy parecido al que se ha obtenido en la actividad 28. La diferencia es que si llega un 1 estando en el estado *ZUUZ*, vamos al estado *ZU*, es decir, el circuito reconoce que ha llegado la subsecuencia 01. Por tanto, descubrimos que el 0 que nos ha llevado hasta el estado *ZUUZ* se considera como el primero de una secuencia nueva. Así pues, el circuito reconoce la secuencia 0110, pero permite el solapamiento entre secuencias consecutivas.

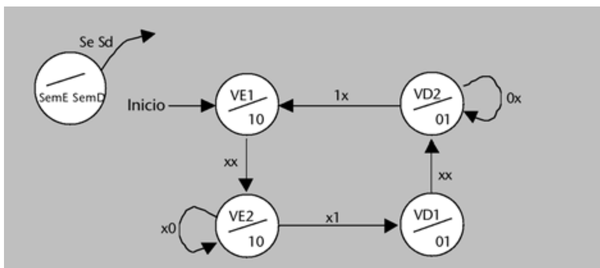
**30.**

Las entradas del circuito son *Sd* y *Se*, y las salidas, *SemD* y *SemE*. En cada momento puede haber sólo un semáforo en verde y, por tanto, siempre se cumplirá que *SemD* = *SemE'*.

Cuando un semáforo se pone en verde debe permanecer en este estado por lo menos dos ciclos de reloj; por tanto, habrá dos estados (*VD1* y *VD2*) en los que *SemD* = 1 y dos estados (*VE1* y *VE2*) en los que *SemE* = 1.

Cuando un semáforo ya ha estado en verde dos ciclos, no se volverá a poner en rojo hasta que lleguen coches por el otro lado.

Dado que inicialmente el semáforo izquierdo se tiene que mantener en verde durante dos ciclos, el estado inicial es *VE1*. El grafo, pues, es el siguiente:

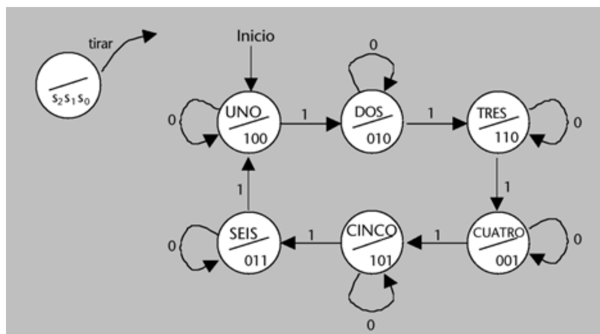



**31.**

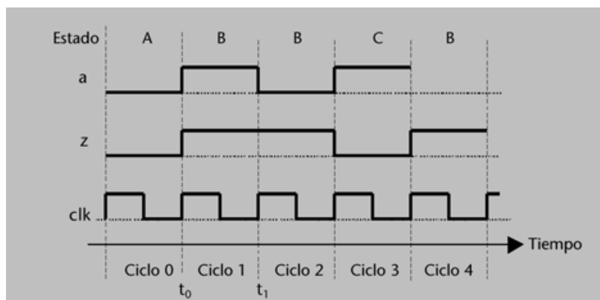
Veamos cómo hay que iluminar los puntos del dado para formar las diferentes combinaciones:

| Combinación | Puntos iluminados | Salidas | | |
|---|---|---|---|---|
| | | $s_2$ | $s_1$ | $s_0$ |
| 1 | d | 1 | 0 | 0 |
| 2 | c, e | 0 | 1 | 0 |
| 3 | c, d, e | 1 | 1 | 0 |
| 4 | a, b, f, g | 0 | 0 | 1 |
| 5 | a, b, d, f, g | 1 | 0 | 1 |
| 6 | a, b, c, e, f, g, | 0 | 1 | 1 |

El grafo de estados es el siguiente (el estado inicial es U):

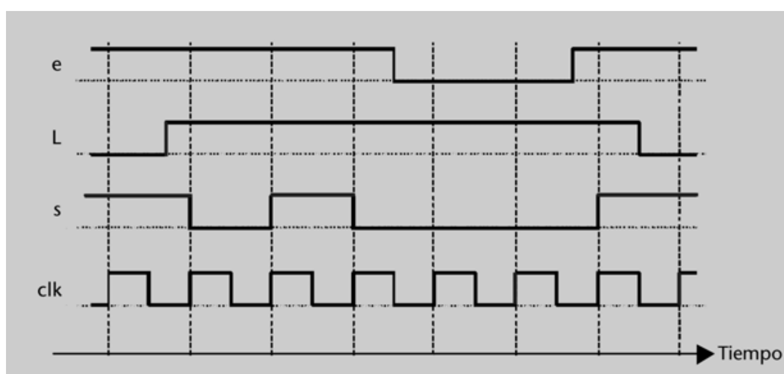



**32.**
En el ciclo 0, el circuito está en el estado *A*. En el ciclo 1, pasa al estado *B*. Por tanto, en el instante $t_0$ la entrada tenía que valer 0. Dado que suponemos que la entrada sólo cambia de valor en los flancos, obtenemos que $a = 0$ durante todo el ciclo 0.
Para que el circuito continúe en el estado *B* en el ciclo 2, es preciso que en el instante $t_1$ (y, por tanto, durante todo el ciclo 1), la entrada valga 1. Durante el ciclo 2 la entrada tiene que valer 0, para que el circuito pase al estado *C* en el ciclo 3. No disponemos de ninguna información que nos permita determinar cuánto vale la entrada durante el ciclo 4.
El valor de la señal de salida viene determinada por el estado en que se encuentra el circuito: 0 mientras está en el estado *A* o en el *C*, y 1 mientras está en el estado *B*.
El cronograma completo se muestra a continuación:



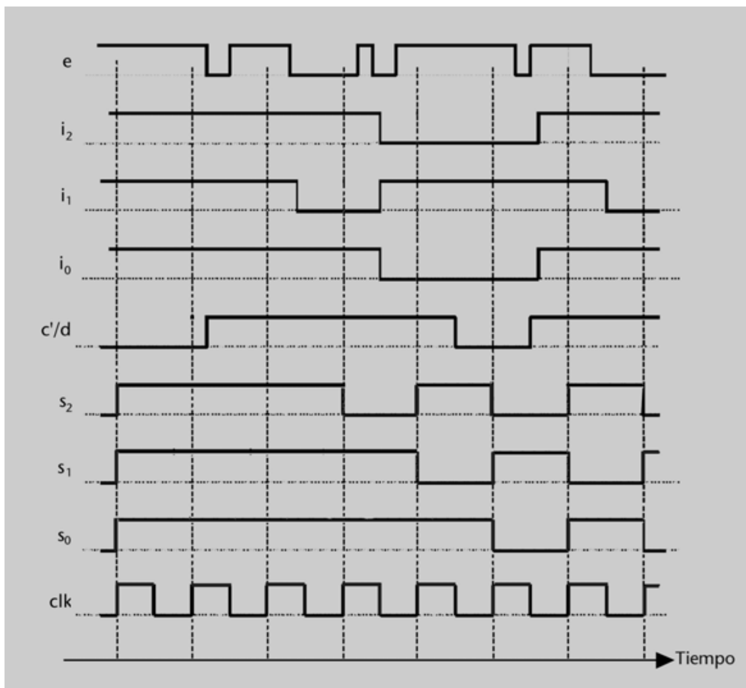

## Ejercicios de autoevaluación

**1.** La señal *e* está conectada a una puerta AND con la salida conectada a la entrada *D* del biestable. Por tanto, siempre que *e* valga 0 llegará un 0 a esta entrada. Podríamos decir que cumple el papel de una señal de *reset* síncrono. Mientras *e* vale 1, el biestable invierte su valor en cada ciclo. La señal *L*, conectada a la entrada *load*, inhibe la carga del biestable cuando se pone a 0.




**2.** Como se puede observar en el siguiente cronograma, cuando la entrada *c'/d* vale 0, las salidas del circuito ($s_2$, $s_1$ y $s_0$, que corresponden al valor guardado en cada uno de los biestables) toman el valor de las entradas $i_2$, $i_1$ $i_0$, respectivamente.

Por otro lado, cuando la entrada *c'/d* vale 1, las salidas del circuito toman el valor del biestable de la izquierda. Por tanto, se produce un desplazamiento de los bits almacenados una posición hacia la derecha. El bit más a la izquierda, $s_2$, se carga con el valor que haya en la entrada *e*.




**3.**

**a)** El bit *c* vale 0 si $A < B$ y 1 si $A \geq B$.
Este bit *c* está conectado a la salida $f_{out}$ del bloque B1 situado a la izquierda de todos. Veamos cuánto tiene que valer esta salida $f_{out}$:

- Si $a_3 = 0$ y $b_3 = 1$, entonces $A < B$, de forma que *c* (y por tanto, $f_{out}$) tiene que valer 0.
- Si $a_3 = 1$ y $b_3 = 0$, entonces $A > B$, con lo que *c* (y por tanto, $f_{out}$) tiene que valer 1.
- Si $a_3 = b_3$, entonces hay que saber cuánto valen los bits de menos peso de *A* y *B* para decidir el valor que tendrá que tomar *c*.

Fijémonos en el valor del punto *d* del circuito. Ésta sería la salida del circuito B2 en el caso de que los números fuesen de tres bits en lugar de cuatro. Por tanto, $d = 0$ si $[a_2\ a_1\ a_0] < [b_2\ b_1\ b_0]$ y 1 en el caso contrario. Ésta es la información que falta para decidir el valor de *c* en el caso de que $a_3 = b_3$. Por tanto, debido a que *d* está conectada a la entrada $f_{in}$ del último bloque B1, obtenemos que la tabla de verdad del bloque B1 situado más a la izquierda es la siguiente (en general, la tabla de verdad de un bloque B1 cualquiera será la misma, cambiando sólo $a_3$ y $b_3$ por $a_i$ y $b_i$):

| $a_3$ | $b_3$ | $f_{in}$ | $f_{out}$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $A < B$, ya que $a_3 = b_3$ i $[a_2\ a_1\ a_0] < [b_2\ b_1\ b_0]$ |
| 0 | 0 | 1 | 1 | $A \geq B$, ya que $a_3 = b_3$ i $[a_2\ a_1\ a_0] \geq [b_2\ b_1\ b_0]$ |
| 0 | 1 | 0 | 0 | $A < B$, ya que $a_3 < b_3$ |
| 0 | 1 | 1 | 0 | " |
| 1 | 0 | 0 | 1 | $A \geq B$, ya que $a_3 > b_3$ |
| 1 | 0 | 1 | 1 | " |
| 1 | 1 | 0 | 0 | $A < B$, ya que $a_3 = b_3$ i $[a_2\ a_1\ a_0] < [b_2\ b_1\ b_0]$ |
| 1 | 1 | 1 | 1 | $A \geq B$, ya que $a_3 = b_3$ i $[a_2\ a_1\ a_0] \geq [b_2\ b_1\ b_0]$ |

Los bits de peso –1 no existen, pero el bloque B1 de la derecha se tiene que comportar "como si fuesen iguales", es decir, como si $a_{-1} = b_{-1}$. Esto se consigue conectado un 1 a su entrada $f_{in}$.

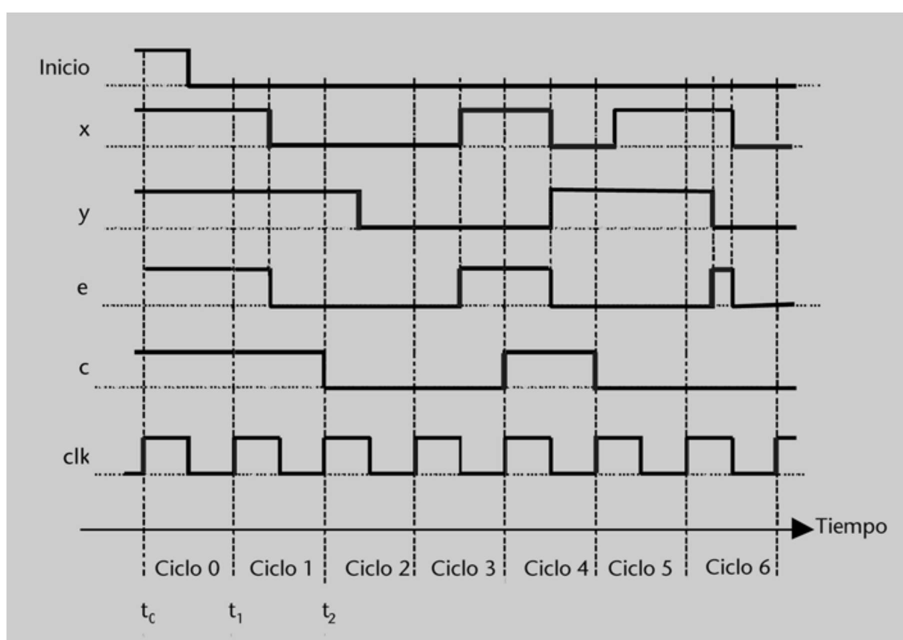
**b)** En el apartado **a)** hemos visto que cada bloque B1 hace la comparación de dos bits *A* y *B*. Esta comparación consulta, cuando no puede decidir sólo con los bits $a_i$ y $b_i$, el resultado de la comparación de los bits de menos peso.

El circuito B3 compara dos bits de entrada, $x$ e $y$, y almacena el resultado de la comparación en un biestable. El bloque B1 consultará este resultado cuando compare los bits $x$ e $y$ en el siguiente flanco de reloj, ya que la salida del biestable está conectada a la entrada $f_{in}$.

Si comparamos este circuito con el del apartado **a)**, podemos ver que pueden cumplir la misma función, si en el primer ciclo de reloj se conectan $[a_0\ b_0]$ a los puntos $[x\ y]$, en el segundo ciclo se conectan $[a_1\ b_1]$, y así sucesivamente. La señal *Inicio* se puede usar para cargar un 1 en el biestable cuando empieza a funcionar el circuito, es decir, puede hacer el papel del 1 que se conecta a la entrada $f_{in}$ del bloque B1 que está a la derecha del todo en el circuito B2.

La diferencia entre los circuitos B2 y B3 es que el primero es combinacional, mientras que el segundo es secuencial. En el circuito B2, todos los bits de los números $A$ y $B$ se comparan a la vez, y el resultado de la comparación estará disponible al instante (para ser más precisos, habrá que esperar sólo el tiempo de retraso de las puertas que forman los bloques B1). En cambio, en el circuito B3, los bits de $A$ y $B$ se comparan pareja por pareja, de forma secuencial en el tiempo. El circuito debe tener un biestable para recordar en cada momento el resultado de la comparación del par anterior. El resultado final de la comparación estará disponible cuatro ciclos después de que el circuito empiece a funcionar.

**c)** El cronograma se muestra a continuación. Observamos que el punto $e$ cambia de valor de forma asíncrona, de acuerdo con las variaciones en $x$ e $y$.




Durante el ciclo 0, $x = y = 1$. Por tanto, $a_0 = b_0 = 1$. Durante el ciclo 1, $x$ cambia de valor. El valor que se guarde en el biestable depende del valor de $x$, $y$ y $e$ al final del ciclo 1 (en el instante $t_2$) y, por tanto, diremos que $a_1 = 0$. Si seguimos el mismo razonamiento en todos los casos, obtenemos que los números que se han comparado son los siguientes:

$$A = 0101001,\ B = 0110011.$$

Se cumple que $A < B$. Podemos comprobar que, efectivamente, la salida $c$ del circuito vale 0.

**4.**

**a)** El circuito tiene que saber si es de día o de noche para regular adecuadamente la duración de cada luz. Por tanto, necesita una señal de entrada, que llamaremos *d'/n*, y supondremos que vale 0 cuando es de día y 1 cuando es de noche.

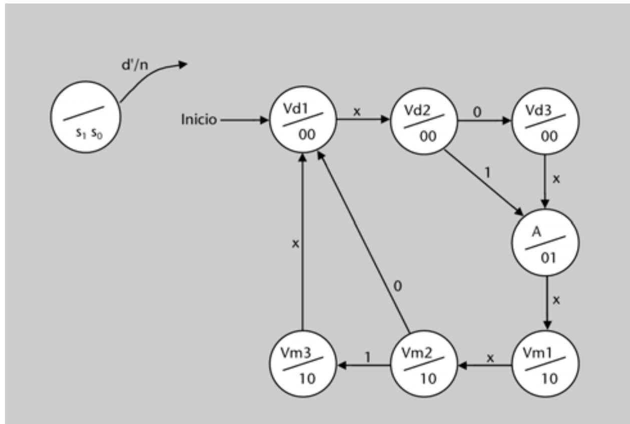
Las salidas tienen que indicar qué luz está encendida en cada momento. Por ejemplo, el circuito podría generar 3 salidas, una asociada a cada luz. Otra posibilidad es que genere sólo dos señales de salida, que controlen el semáforo de la siguiente forma:

| $s_1$ | $s_0$ | Semáforo |
|---|---|---|
| 0 | 0 | Verde |
| 0 | 1 | Amarillo |
| 1 | 0 | Rojo |

Tomaremos esta segunda posibilidad.

**b)** El grafo de estados se muestra a continuación. Hemos identificado para *Vdi* los estados durante los cuales el semáforo está en verde, para *A* el estado durante el cual el semáforo está en amarillo, y para *Vmi* los estados durante los cuales el semáforo está en rojo. Hemos supuesto que el estado inicial es *Vd1*, pero el enunciado no nos indicaba nada al respecto, por lo que podíamos haber tomado cualquier estado como inicial.

Observamos que, durante el día, sólo se recorren dos de los estados en que el semáforo está en rojo, y por la noche sólo se recorren dos de los estados en que el semáforo está en verde.

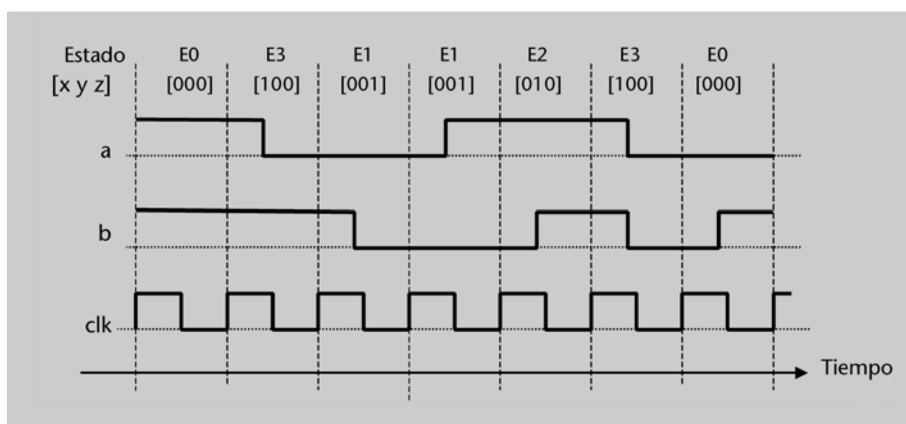



**5.**
**a)** Las tablas se muestran a continuación:

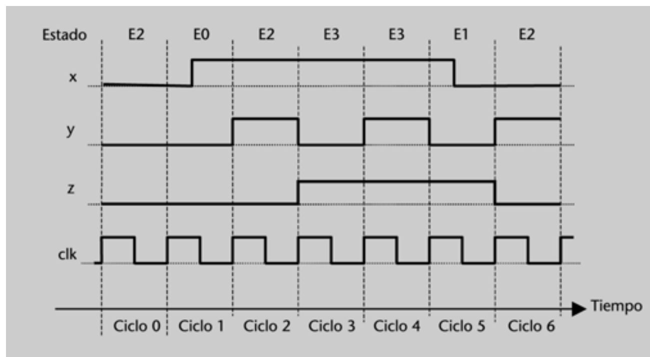
| Tabla de salidas | |
|---|---|
| **Estado** | **x y z** |
| E0 | 0 0 0 |
| E1 | 0 0 1 |
| E2 | 0 1 0 |
| E3 | 1 0 0 |

| Tabla de transiciones | | | |
|---|---|---|---|
| **Estado** | **a** | **b** | **Estado$^+$** |
| E0 | 0 | 0 | E0 |
| E0 | 0 | 1 | E1 |
| E0 | 1 | 0 | E2 |
| E0 | 1 | 1 | E3 |
| E1 | 0 | 0 | E1 |
| E1 | 0 | 1 | x |
| E1 | 1 | 0 | E2 |
| E1 | 1 | 1 | E2 |
| E2 | 0 | 0 | E1 |
| E2 | 0 | 1 | E3 |
| E2 | 1 | 0 | E1 |
| E2 | 1 | 1 | E3 |
| E3 | 0 | 0 | E0 |
| E3 | 0 | 1 | E1 |
| E3 | 1 | 0 | x |
| E3 | 1 | 1 | x |

**b)** El cronograma se muestra a continuación:

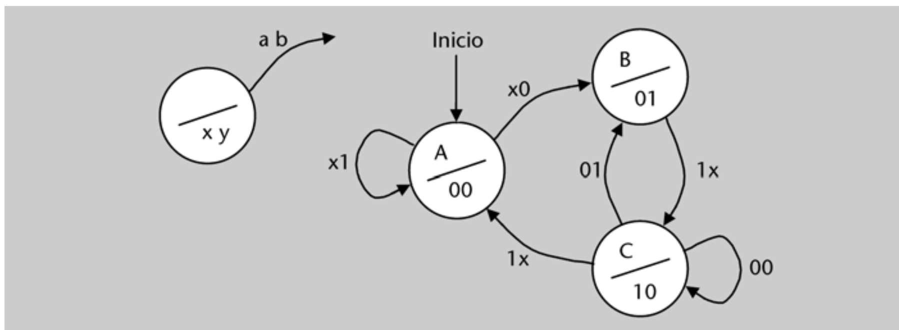


**6.**
El cronograma completo se muestra a continuación:

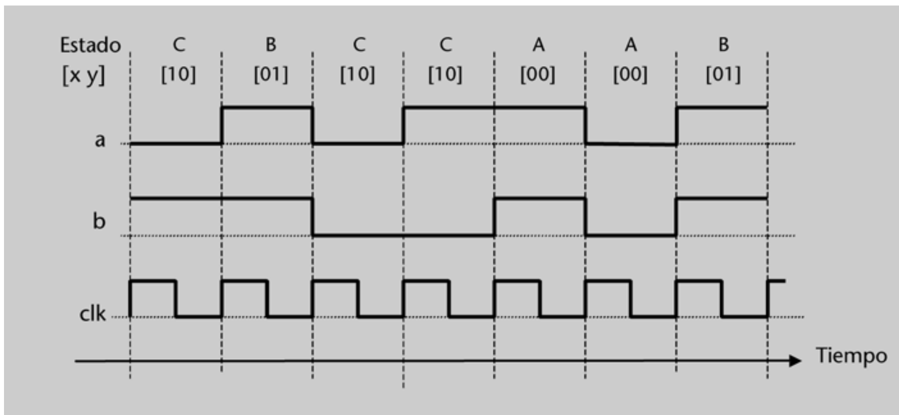



Recordad que en los circuitos secuenciales en el momento del flanco ascendente se debe coger el valor que tienen las entradas a la izquierda del flanco. Por ejemplo, al inicio del ciclo 2, *x* vale 1 y *y* vale 0.

**7.**
El grafo se muestra a continuación.




**b)** El cronograma se muestra a continuación:

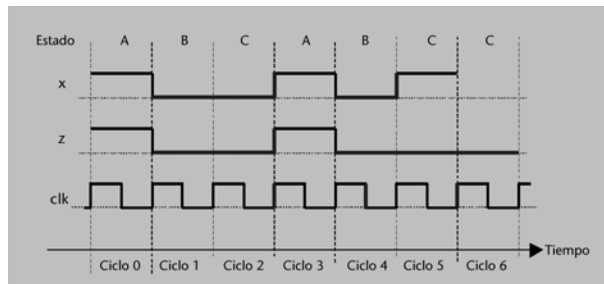



**8.**
Completaremos el cronograma por pasos:

- Durante el ciclo 1 el circuito está en el estado *B* y, por tanto, la salida vale 0. Sólo se puede llegar al estado *B* desde el *A* con entrada 1. Por tanto, deducimos que durante el ciclo 0 el circuito estaba en el estado *A* (por tanto, la salida valía 1) y que la entrada valía 1.
- Durante el ciclo 1, estamos en el estado *B* y la entrada es 0; por tanto, en el siguiente ciclo vamos al estado *C* con salida 0.
- Durante el ciclo 3, la salida vale 1. El único estado en el que la salida vale 1 es el *A*. Para llegar al estado *A* a partir del *C*, se tiene que cumplir que durante el ciclo 2, la entrada valga 0.
- Durante el ciclo 4, la salida vale 0. Si tenemos en cuenta que venimos del estado *A*, deducimos que en un ciclo 4 estamos en el estado *B* (ya que se puede ir al *B* desde el *A*); por tanto, durante el ciclo 3 la entrada valía 1.
- Durante el ciclo 4 estamos en el estado *B* y la entrada vale 0; por tanto, en el ciclo 5 estaremos en el estado *C*, y la salida valdrá 0.

- En el ciclo 6 vamos a un estado en el que la salida vale 0. Si tenemos en cuenta que durante el ciclo 5 estábamos en el estado *C*, deducimos que nos hemos quedado en el mismo estado; por tanto, la entrada durante el ciclo 5 tenía que valer 1.
- No tenemos ninguna información que nos permita saber cuánto vale la entrada durante el ciclo 6.

El cronograma completo se muestra a continuación:




**9.**
Inicialmente, el circuito está en el estado *E0*, y se queda mientras *p_ab* = 0. Si *p_ab* vale 1, el circuito pasa al estado *E1*. Deducimos, pues, que el estado *E1* indica que la puerta de la caja fuerte ha estado abierta durante 30 segundos (un ciclo de reloj). Podemos seguir el mismo razonamiento para ver que el estado *E2* señala que la caja ha estado abierta durante un minuto, y así sucesivamente hasta el estado *E4*, que indica que la caja ha estado abierta durante dos minutos seguidos. La salida *alarma* vale 1 sólo en el estado *E4*.
A partir de las transiciones, concluimos que la señal de alarma se activa cuando la caja permanece abierta durante dos minutos o más, y se desactiva un ciclo después de que se cierre la puerta de la caja fuerte.

# Bibliografía

**Gajsky, D.D.** (1997). *Principios de diseño Digital*. Prentice-Hall.

**Hermida, R.; Corral, A. del; Pastor, E.; Sánchez, F.** (1998). *Fundamentos de Computadores*. Madrid: Síntesis.

# Estructura básica de un computador

## El procesador como generalización de las máquinas algorítmicas

Lluís Ribas i Xirgo

PID_00163601

Universitat Oberta de Catalunya

www.uoc.edu

# Índice

**Introducción** .......... 5

**Objetivos** .......... 6

**1. Máquinas de estados** .......... 7
1.1. Máquinas de estados finitos como controladores .......... 8
1.1.1. Procedimiento de materialización de controladores con circuitos secuenciales .......... 10
1.2. Máquinas de estados finitos extendidas .......... 16
1.3. Máquinas de estados-programa .......... 26
1.4. Máquinas de estados algorítmicas .......... 41

**2. Máquinas algorítmicas** .......... 52
2.1. Esquemas de cálculo .......... 52
2.2. Esquemas de cálculo segmentados .......... 55
2.3. Esquemas de cálculo con recursos compartidos .......... 56
2.4. Materialización de esquemas de cálculo .......... 58
2.5. Representación de máquinas algorítmicas .......... 60
2.6. Materialización de máquinas algorítmicas .......... 62
2.7. Caso de estudio .......... 65

**3. Arquitectura básica de un computador** .......... 70
3.1. Máquinas algorítmicas generalizables .......... 71
3.1.1. Ejemplo de máquina algorítmica general .......... 72
3.2. Máquina elemental .......... 79
3.2.1. Máquinas algorítmicas de unidades de control .......... 81
3.2.2. Máquinas algorítmicas microprogramadas .......... 83
3.2.3. Una máquina con arquitectura de Von Neumann .......... 86
3.3. Procesadores .......... 91
3.3.1. Microarquitecturas .......... 91
3.3.2. Microarquitecturas con *pipelines* .......... 92
3.3.3. Microarquitecturas paralelas .......... 93
3.3.4. Microarquitecturas con CPU y memoria diferenciadas .......... 94
3.3.5. Procesadores de propósito general .......... 96
3.3.6. Procesadores de propósito específico .......... 96
3.4. Computadores .......... 98
3.4.1. Arquitectura básica .......... 98
3.4.2. Arquitecturas orientadas a aplicaciones específicas .......... 102

**Resumen** .......... 104

**Ejercicios de autoevaluación** .......... 107

**Solucionario** .................................................................................... 112

**Glosario** .................................................................................... 133

**Bibliografía** .................................................................................... 137

# Introducción

En el módulo anterior se ha visto que los circuitos secuenciales sirven para detectar el orden temporal de una serie de sucesos (es decir, secuencias de bits) y que también, a partir de sus salidas, pueden controlar todo tipo de sistemas. Como su funcionalidad se puede representar con un grafo de estados, a menudo se habla de *máquinas de estados*. Así pues, se ha visto cómo construir circuitos secuenciales a partir de las representaciones de las máquinas de estados correspondientes.

De hecho, la mayoría de los sistemas digitales, incluidos los computadores, son sistemas secuenciales complejos, compuestos, finalmente, por una multitud de máquinas de estados y de elementos de procesamiento de datos. O dicho de otra manera, los sistemas secuenciales complejos están constituidos por un conjunto de **unidades de control** y de **unidades de procesamiento**, que materializan las máquinas de estados y los bloques de procesamiento de datos, respectivamente.

En este módulo aprenderemos a enfrentarnos con el problema de analizar y sintetizar circuitos secuenciales de manera sistemática. De hecho, muchos de los problemas que se proponen en circuitos secuenciales del tipo "cuando se cumpla una condición sobre unos determinados valores de entrada, estando en un estado determinado, se dará una salida que responde a unos cálculos concretos y que puede corresponder a un estado diferente del primero" se pueden resolver mejor si se piensa solo en términos de máquinas de estados, que si se intenta efectuar el diseño de las unidades de procesamiento y de control por separado. Lo mismo sucede en casos un poco más complejos, donde la frase anterior empieza con "si se cumple" o "mientras se cumpla", o si los cálculos implican muchos pasos.

Finalmente, eso ayudará a comprender cómo se construyen y cómo trabajan los procesadores, que son el componente principal de todo computador.

El módulo acaba mostrando las arquitecturas generales de los computadores, de manera que sirva de base para comprender el funcionamiento de estas máquinas.

# Objetivos

La misión de este módulo es que se aprendan los conceptos fundamentales para el análisis y la síntesis de los distintos componentes de un computador, visto como sistema digital secuencial complejo. Se enumeran, a continuación, los objetivos particulares que se deben alcanzar después del estudio de este módulo.

1. Tener conocimiento de los distintos modelos de máquinas de estados y de las arquitecturas de controlador con camino de datos.

2. Haber adquirido una experiencia básica en la elección del modelo de máquina de estados más adecuado para la resolución de un problema concreto.

3. Saber analizar circuitos secuenciales y extraer el modelo correspondiente.

4. Ser capaz de diseñar circuitos secuenciales a partir de grafos de transiciones de estados.

5. Conocer el proceso de diseño de máquinas algorítmicas y entender los criterios que lo afectan.

6. Tener la capacidad de diseñar máquinas algorítmicas a partir de programas sencillos.

7. Haber aprendido el funcionamiento de los procesadores como máquinas algorítmicas de interpretación de programas.

8. Conocer la arquitectura básica de los procesadores.

9. Tener habilidad para analizar las diferentes opciones de materialización de los procesadores.

10. Haber adquirido nociones básicas de la arquitectura de los computadores.

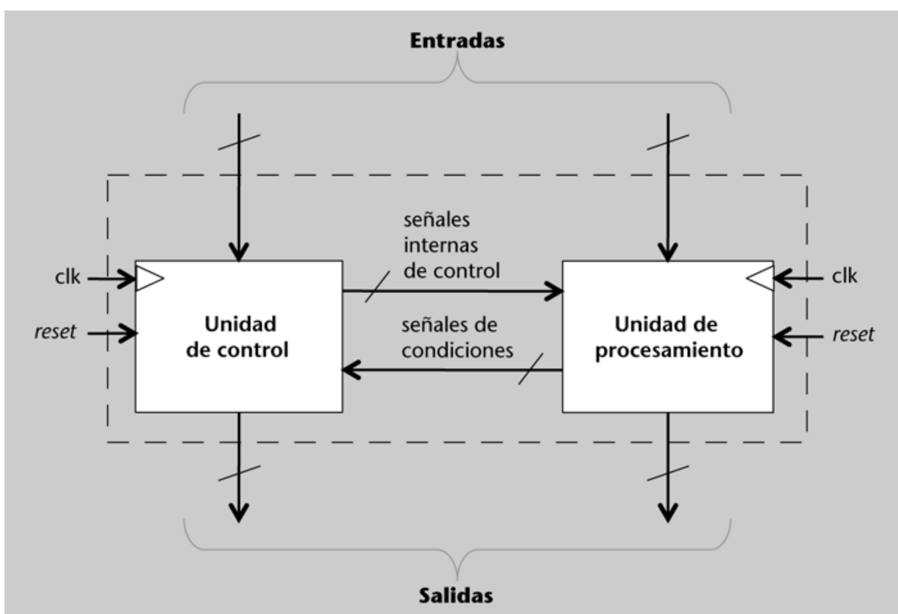
# 1. Máquinas de estados

Los circuitos secuenciales son, de hecho, máquinas de estados cuyo comportamiento se puede representar con un grafo de transición de estados. Si se asocia a cada estado la realización de una determinada operación, entonces las máquinas de estados son una manera de ordenar la realización de un conjunto de operaciones. Este orden queda determinado por una secuencia de entradas y el estado inicial de la máquina correspondiente.

Así pues, los circuitos secuenciales se pueden organizar en dos partes: la que se ocupa de las transiciones de estados y que implementan las funciones de excitación de la máquina de estados correspondiente, y la que se ocupa de realizar las operaciones con los datos, tanto para determinar condiciones de transición como para calcular resultados de salida. Como ya se ha comentado en la introducción, la primera se denomina **unidad de control** y la segunda, **unidad de procesamiento**.

La figura 1 ilustra esta organización. La unidad de control es una máquina de estados que recibe información del exterior por medio de señales de entrada y también de la unidad de procesamiento, en forma de indicadores de condiciones. Cada estado de la máquina correspondiente tiene asociadas unas salidas que pueden ser directamente al exterior o hacia la unidad de procesamiento para que efectúe unos cálculos determinados. Estos cálculos se realizan con datos del exterior y también internos, ya que la unidad de procesamiento dispone de elementos de memoria. Los resultados de estos cálculos pueden ser observables desde el exterior y, por ello, esta unidad también tiene señales de salida.

Figura 1. Organización de un circuito secuencial



En el módulo anterior se ha explicado cómo funcionan los circuitos secuenciales que materializan una máquina de estados con poco o nada de procesamiento. En este apartado se tratará de cómo relacionar los estados con las operaciones, de manera que la representación de la máquina de estados incluya también toda la información necesaria para la implementación de la unidad de procesamiento.

### 1.1. Máquinas de estados finitos como controladores

Las **máquinas de estados finitos** (o FSM, del inglés *finite state machines*) son un modelo de representación del comportamiento de circuitos (y de programas) muy adecuado para los controladores.

Un **controlador** es una entidad con capacidad de actuar sobre otra para llevarla a un estado determinado.

El regulador de potencia de un calefactor es un controlador de la intensidad del radiador. Cada posición del regulador (es común que sean las siguientes: 0 para apagado, 1 para media potencia y 2 para potencia máxima) sería un estado del controlador y también un objetivo para la parte controlada. Es decir, se puede interpretar que la posición 1 del regulador significa que hay que llevar el radiador al estado de consumo de una intensidad media de corriente. El mismo ejemplo ilustra la diferencia con una máquina de estados no finitos: si el regulador consistiera en una rueda que se puede girar hasta situarse en cualquier posición entre 0 y 2, entonces sería necesario un número (teóricamente) infinito de estados.

Otro ejemplo de controlador es el del mecanismo de llenado de agua del depósito de una taza de inodoro. En este caso, el "sistema" que se debe controlar tiene dos estados: el depósito puede estar lleno (LLENO) o puede no estarlo (NO_LLENO).

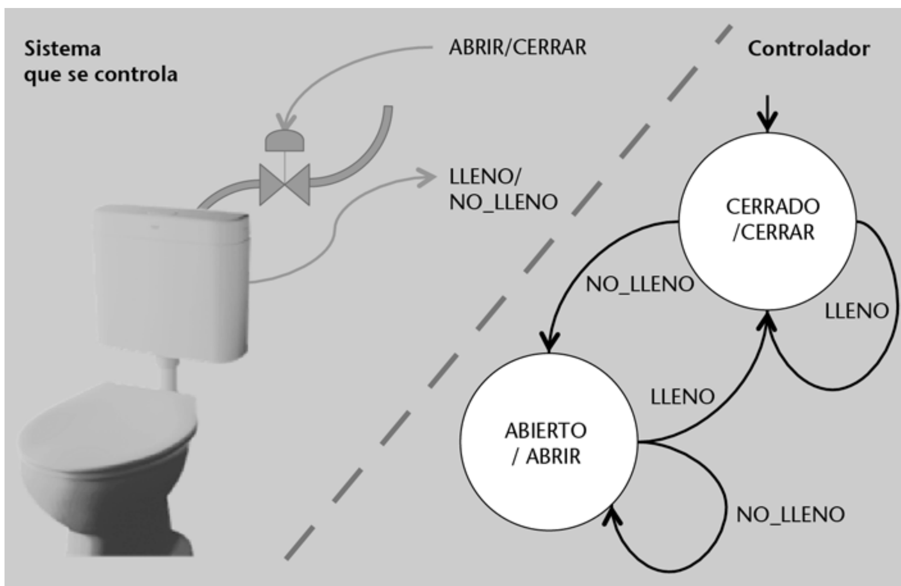
El mecanismo de control de llenado de agua se ocupa de llevar el sistema al estado de LLENO de manera autónoma. En otras palabras, el controlador tiene una referencia interna del estado al que ha de llevar el sistema que controla. Para hacerlo, debe detectar el estado y, con este dato de entrada, determinar qué acción debe realizar: abrir el grifo de llenado (ABRIR) o cerrarlo (CERRAR). Además, hay que tener presente que el controlador deberá mantener el grifo abierto mientras el depósito no esté lleno y cerrado siempre que esté lleno. Esto implica que haya de "recordar" en qué estado se encuentra: con el grifo abierto (ABIERTO) o cerrado (CERRADO).

El funcionamiento de los controladores se puede representar con un grafo de estados y también con tablas de transiciones y de salidas. En este sentido, hay que tener presente que los **estados captados** de los sistemas que controlan

constituyen las entradas de la máquina de estados correspondiente y que las **acciones** son las salidas vinculadas a los estados de los mismos controladores.

Siguiendo con el ejemplo, los estados captados son LLENO/NO_LLENO, las acciones son CERRAR y ABRIR y el objetivo para el sistema controlado, que esté LLENO. El estado inicial del controlador de llenado de la cisterna debe ser CERRADO, que está asociado a la acción de CERRAR. De esta manera, al comenzar a funcionar el controlador, queda garantizado que no habrá desbordamiento del depósito. Si ya está LLENO, debe quedarse en el mismo estado, pero si se detecta que está NO_LLENO, entonces se debe llenar, lo que se consigue pasando al estado de ABIERTO, en el que se abre el grifo. El grifo se mantiene abierto hasta que se detecte que el depósito está LLENO. En este momento, se debe CERRAR, pasando al estado de CERRADO.

Figura 2. Esquema del sistema con el grafo de estados del controlador




La tabla de transiciones de estados es la siguiente:

| Estado del controlador | Estado del depósito | Estado siguiente del controlador |
|---|---|---|
| **Estado actual** | **Entrada** | **Estado futuro** |
| CERRADO | NO_LLENO | ABIERTO |
| CERRADO | LLENO | CERRADO |
| ABIERTO | NO_LLENO | ABIERTO |
| ABIERTO | LLENO | CERRADO |

Y la tabla de salidas es la siguiente:

| Estado del controlador | Acción |
|---|---|
| **Estado actual** | **Salida** |
| ABIERTO | ABRIR |
| CERRADO | CERRAR |

La **materialización** del controlador de llenado del depósito de agua de una taza de inodoro se puede resolver muy eficazmente sin ningún circuito secuencial: la mayoría de las cisternas llevan una válvula de admisión de agua controlada por un mecanismo con una boya que flota. No obstante, aquí se hará con un circuito digital que implemente la máquina de estados correspondiente, un sensor de nivel y una electroválvula. Para ello, es necesario que se codifique toda la información en binario.

En este caso, la observación del estado del sistema (la cisterna del inodoro) se lleva a cabo captando los datos del sensor, que son binarios: LLENO (1) o NO_LLENO (0). Las actuaciones del controlador se realizan por medio de la activación de la electroválvula, que debe ser una que normalmente esté cerrada (0) y que se mantenga abierta mientras esté activada (1). El estado del controlador también se debe codificar en binario. Por ejemplo, haciendo que coincida con la codificación binaria de la acción de salida. Así pues, para construir el circuito secuencial del controlador del depósito, se debe realizar una codificación binaria de todos los datos, tal como se muestra en la tabla siguiente.

| **Estado del depósito** | | **Estado del controlador** | | **Acción** | |
|---|---|---|---|---|---|
| **Identificación** | **Código** | **Controlador** | **Código** | **Identificación** | **Código** |
| NO_LLENO | 0 | CERRADO | 0 | CERRAR | 0 |
| LLENO | 1 | ABIERTO | 1 | ABRIR | 1 |

Como se ha visto en el ejemplo anterior, el estado percibido del sistema que se controla está constituido por los diferentes datos de entrada del controlador. Para evitar confusiones en las denominaciones de los dos tipos de estados, al estado percibido se le hará referencia como **entradas**.

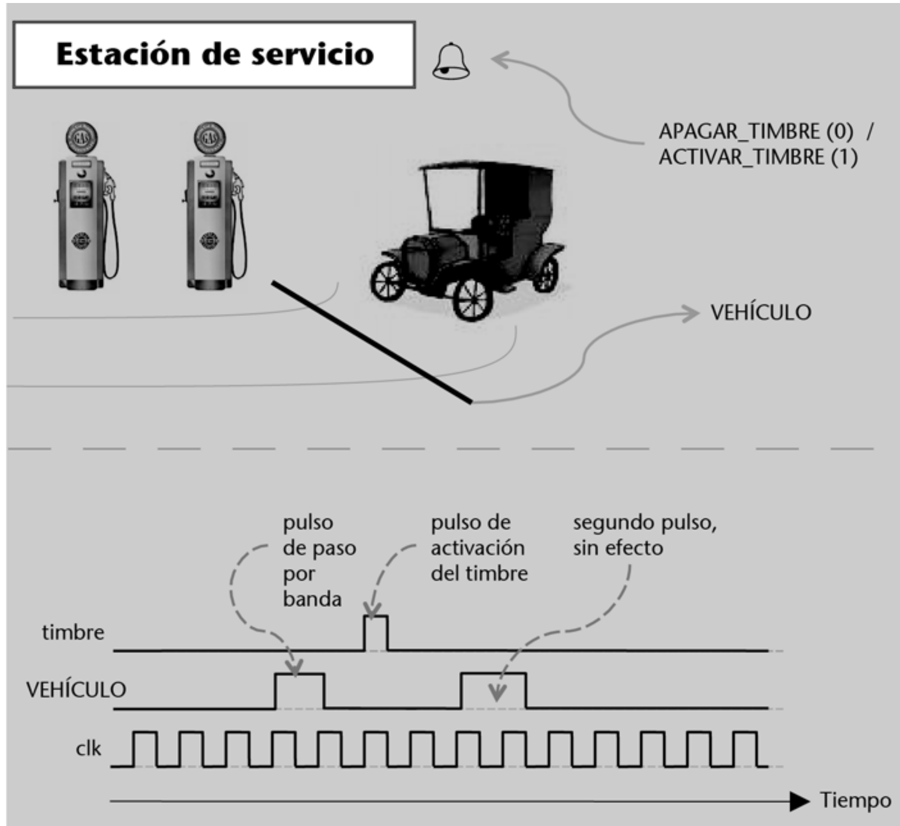
### 1.1.1. Procedimiento de materialización de controladores con circuitos secuenciales

Para diseñar un circuito secuencial que implemente una máquina de estados correspondiente a un controlador, se deben seguir los pasos que se describirán a continuación. Para ilustrarlos, se ejemplificarán con el caso de un controlador que hace sonar un timbre de aviso de entrada de vehículos en una estación de servicio (figura 3).

**1) De acuerdo con la funcionalidad que se tenga que implementar, decidir qué acciones se deben aplicar sobre el sistema que se controla.**

En este caso, solo hay dos acciones: hacer sonar el timbre (ACTIVAR_TIMBRE) o dejarlo apagado (APAGAR_TIMBRE). Tal como se muestra en la parte inferior de la figura 3, para activar el timbre, es suficiente con enviar un pulso a 1 al aparato correspondiente. Por simplicidad, se supone que solo debe durar un ciclo de reloj.

Figura 3. Sistema de aviso de entrada de vehículos a una estación de servicio




**2) Determinar qué información se debe obtener del sistema que se controla o, dicho de otro modo, determinar las entradas del controlador.**

Para el caso de ejemplo, el controlador necesita saber si algún vehículo pasa por la vía de entrada. Para ello se puede tener un cable sensible a la presión fijado en el suelo. Si se activa, quiere decir que hay un vehículo pisándolo y que se debe hacer sonar el timbre de aviso. Por lo tanto, habrá una única entrada, VEHÍCULO, que servirá para que el controlador sepa si hay algún vehículo accediendo a la estación. En otras palabras, el controlador puede "ver" la estación en 2 estados (o, si se quiere, "fotos") distintos, según el valor de VEHÍCULO.

Se considera que todos los vehículos tienen dos ejes y que solo se hace sonar el timbre al paso del primero. (Si pasara un vehículo de más de dos ejes, como un camión, el timbre sonaría más de una vez. Con todo, estos casos no se tienen en cuenta para el ejemplo).

**3) Diseñar la máquina de estados del controlador.**

Se trata de construir el grafo de estados que concuerde con el comportamiento que se espera del conjunto: que suene un timbre de alerta cuando un vehículo entre en la estación de servicio.

Para hacerlo, se empieza con un estado inicial de reposo en el que se desconocen las entradas y se decide hacia qué estados debe pasar según todas las posibles combinaciones de entrada. Y para cada uno de estos estados de destino, se hace lo mismo teniendo en cuenta los estados que ya se han creado.

El estado inicial (INACTIVO) tendría asociada la acción de APAGAR_TIMBRE para garantizar que, estuviera como estuviera anteriormente, al poner en marcha el controlador el timbre no sonara. La máquina de estados debe permanecer en este estado hasta que no se detecte el paso de un vehículo, es decir, hasta que VEHÍCULO = 1. En este caso, pasará a un nuevo estado, PULSO_1, para determinar cuándo ha acabado de pasar el primer eje del vehículo y, por lo tanto, cuándo se debe hacer sonar el timbre.

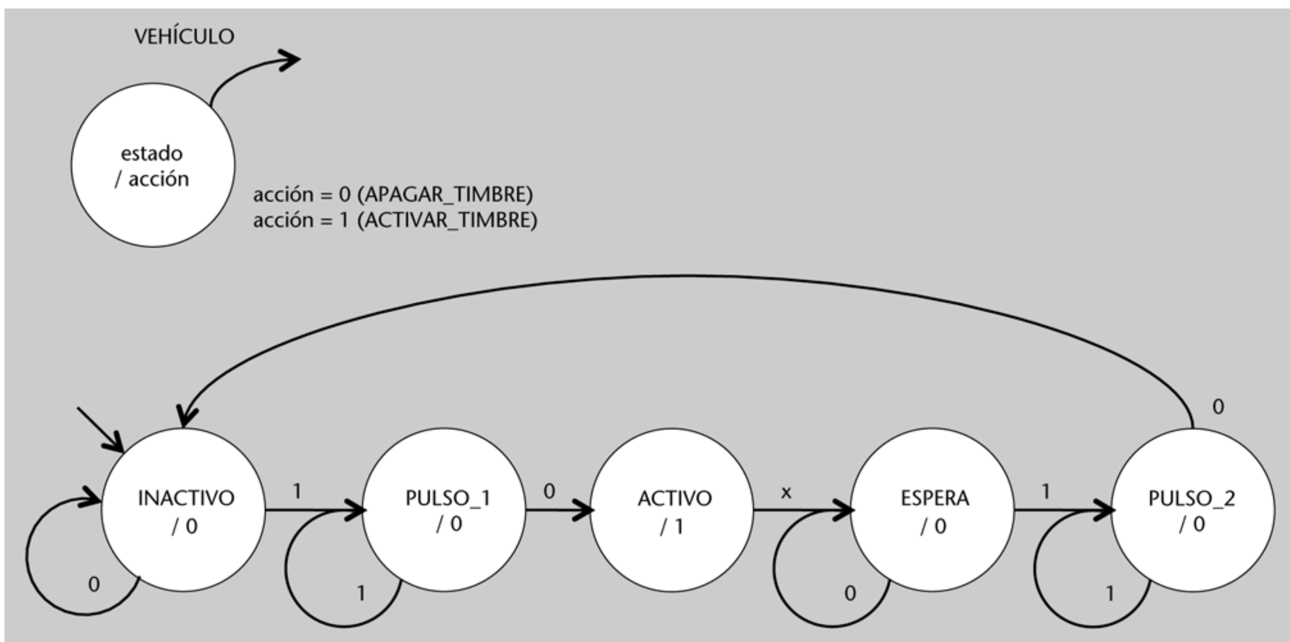
Hay que tener presente que la velocidad de muestreo de la señal VEHÍCULO, es decir, el número de lecturas de este bit por unidad de tiempo, es la del reloj del circuito: mucho mayor que la velocidad de paso de cualquier vehículo. Por lo tanto, la detección de un pulso se realiza al completar una secuencia del tipo 01...10, con un número indefinido de unos.

Así pues, la máquina pasa al estado PULSO_1 después de haber detectado un flanco de subida (paso de 0 a 1) a VEHÍCULO, y no tiene que salir de él hasta que haya un flanco de bajada que indique la finalización del primer pulso. En este momento debe pasar a un estado en el que se active el timbre, ACTIVO.

El estado ACTIVO es el único asociado a la acción de ACTIVAR_TIMBRE. Como esto solo hay que hacerlo durante un ciclo de reloj, con independencia del valor que haya en VEHÍCULO, la máquina de estados pasará a un estado diferente. Este nuevo estado esperará el pulso correspondiente al paso del segundo eje del vehículo. De hecho, serán necesarios dos: el estado de espera de flanco de subida (ESPERA) y, después, el estado de espera de flanco de bajada (PULSO_2) y, por lo tanto, de finalización del pulso.

Una vez acabado el segundo pulso, se tiene que pasar de nuevo al estado inicial, INACTIVO. De esta manera, queda a la espera de que venga otro vehículo.

Figura 4. Grafo de estados del sistema de aviso de entrada de vehículos



Con eso ya se puede construir el diagrama de estados correspondiente. En este caso, como solo hay una señal de entrada, es fácil comprobar que se han tenido en cuenta todos los casos posibles (todas las "fotos" del sistema que se controla) en cada estado.

De cara a la construcción de un controlador robusto, hay que comprobar que no haya secuencias de entrada que provoquen transiciones no deseadas.

Por ejemplo, según el diagrama de estados, el paso de ACTIVO a ESPERA se efectúa sin importar el valor de VEHÍCULO, que puede hacer perder un 1 de un pulso del tipo ...010... Como, en el estado siguiente, VEHÍCULO vuelve a ser 0, el pulso no se lee y la máquina de estados empezará a hacer sonar el timbre, en los vehículos siguientes, al paso del segundo eje y no al del primero. Este caso, sin embargo, es difícil que suceda cuando la velocidad de muestreo es mucho más elevada que la de cambio de valores en VEHÍCULO. No obstante, también se puede alterar el diagrama de estados para que no sea posible. (Se puede hacer como ejercicio voluntario.)

**4) Codificar en binario las entradas, los estados del controlador y las salidas.**

Aunque, en el proceso de diseño de la máquina de estados, las entradas y la salida ya se pueden expresar directamente en binario, todavía queda la tarea de codificación de los estados del controlador. Generalmente se opta por dos políticas:

- tantos bits como estados diferentes y solo uno activo para cada estado (en inglés: *one-hot bit*) o
- numerarlos con números binarios naturales, del 0 al número de estados que haya. Habitualmente, el estado codificado con 0 es el estado inicial, ya que facilita la implementación del *reset* del circuito correspondiente.

Del grafo de estados de la figura 4 se puede deducir la tabla de transiciones siguiente.

| Estado actual | | | | Entrada | Estado siguiente | | | |
|---|---|---|---|---|---|---|---|---|
| Identificación | $q_2$ | $q_1$ | $q_0$ | VEHÍCULO | Identificación | $q_2^+$ | $q_1^+$ | $q_0^+$ |
| INACTIVO | 0 | 0 | 0 | 0 | INACTIVO | 0 | 0 | 0 |
| INACTIVO | 0 | 0 | 0 | 1 | PULSO_1 | 0 | 0 | 1 |
| PULSO_1 | 0 | 0 | 1 | 0 | ACTIVO | 0 | 1 | 0 |
| PULSO_1 | 0 | 0 | 1 | 1 | PULSO_1 | 0 | 0 | 1 |
| ACTIVO | 0 | 1 | 0 | x | ESPERA | 0 | 1 | 1 |
| ESPERA | 0 | 1 | 1 | 0 | ESPERA | 0 | 1 | 1 |
| ESPERA | 0 | 1 | 1 | 1 | PULSO_2 | 1 | 0 | 0 |
| PULSO_2 | 1 | 0 | 0 | 0 | INACTIVO | 0 | 0 | 0 |
| PULSO_2 | 1 | 0 | 0 | 1 | PULSO_2 | 1 | 0 | 0 |

En este caso, la codificación de los estados sigue la numeración binaria.

La tabla de salidas es la siguiente:

| Estado actual | | | | Salida | |
|---|---|---|---|---|---|
| **Identificación** | $q_2$ | $q_1$ | $q_0$ | **Símbolo** | **Timbre** |
| INACTIVO | 0 | 0 | 0 | APAGAR_TIMBRE | 0 |
| PULSO_1 | 0 | 0 | 1 | APAGAR_TIMBRE | 0 |
| ACTIVO | 0 | 1 | 0 | ACTIVAR_TIMBRE | 1 |
| ESPERA | 0 | 1 | 1 | APAGAR_TIMBRE | 0 |
| PULSO_2 | 1 | 0 | 0 | APAGAR_TIMBRE | 0 |

Como apunte final, hay que observar que si el código del estado ESPERA fuera 101 y no 011, la salida sería, directamente, $q_1$.

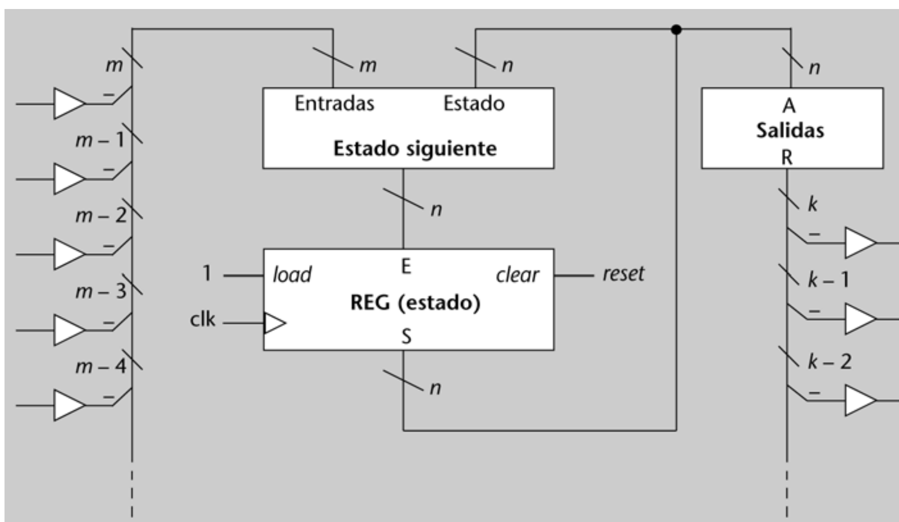
**5) Diseñar los circuitos correspondientes.**

El modelo de construcción de una FSM como controlador que toma señales binarias de entrada y las da de salida se basa en dos elementos: un registro para almacenar el estado y una parte combinacional que calcula el estado siguiente y las salidas, tal como se ve en la figura 5.

**Elementos adaptadores**

En los circuitos a menudo se utilizan unos elementos adaptadores (o *buffers*) que sirven para transmitir señales sin pérdida. El símbolo de estos elementos es como el del inversor sin la bolita. En los esquemas de los circuitos se utilizan también para indicar el sentido de propagación de la señal y, en el caso de este material, para indicar si se trata de señales de entrada o de salida.

Figura 5. Arquitectura de un controlador basado en FSM

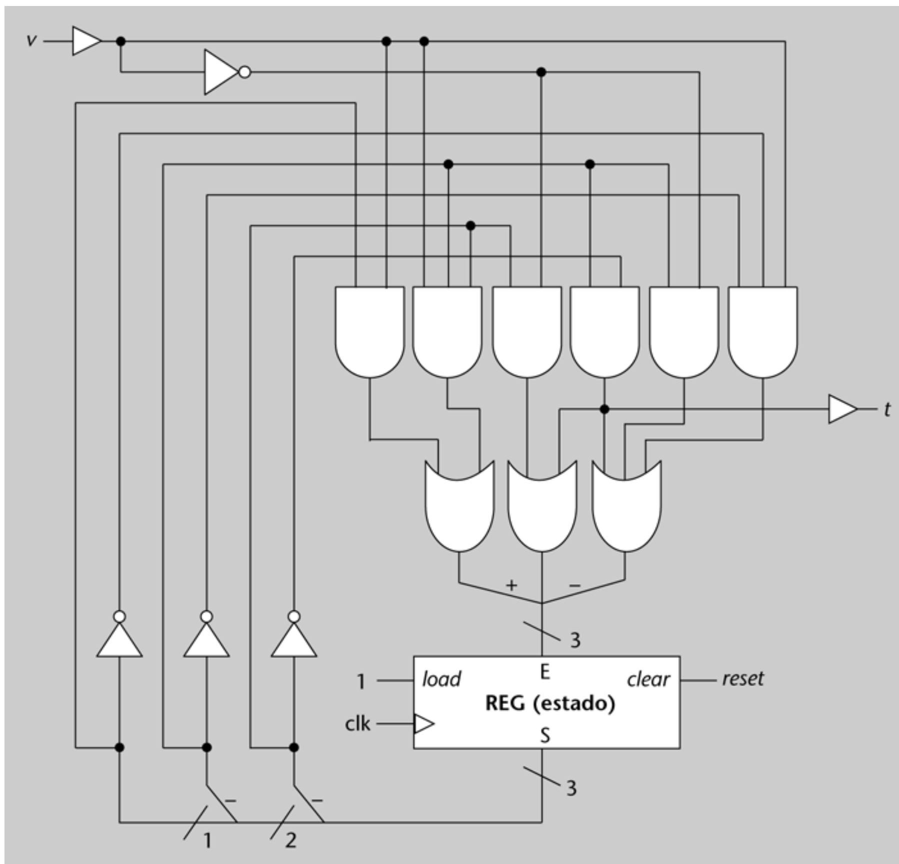



Aunque la parte combinacional se separa en dos: la que calcula el estado siguiente y la que calcula las señales de salida, puede haber elementos comunes que pueden ser compartidos.

Para el controlador del timbre de aviso de entrada de vehículos en la estación de servicio, el circuito del controlador binario es el que se muestra en la figura 6. En el circuito, la señal de entrada *v* es la que indica la detección de paso de eje de vehículos (VEHÍCULO) y *t* (timbre), la de salida, que permite poner en marcha el timbre. Las expresiones de las funciones lógicas de cálculo del estado siguiente comparten un término ($q_1q_0'$) que, además, se puede aprovechar

como función de salida (*t*). Las entradas de los sensores están en el lado izquierdo y las salidas hacia el actuador (el que hace la acción, que es el timbre), en el derecho.

Figura 6. Esquema de la FSM controladora del timbre de aviso de entrada




En este caso, se ha optado por hacerlo con puertas lógicas (se deja, como ejercicio, la comprobación de que el circuito de la figura 6 se corresponde con las funciones lógicas de las tablas de transición y de salida del controlador).

En casos más complejos, es conveniente utilizar bloques combinacionales mayores y también memorias ROM.

### Actividades

**1.** En las comunicaciones en serie entre dispositivos, el emisor puede enviar un pulso a 1 para que el receptor ajuste la velocidad de muestreo de la entrada. Diseñad un circuito “detector de velocidad” que funcione con un reloj a la frecuencia más rápida de comunicación, y que dé, como salida, el valor de división de la frecuencia en la que recibe los pulsos a 1. Los valores de salida pueden ser:

- 00 (no hay detección),
- 01 (ninguna división, el pulso a 1 dura lo mismo que un pulso de reloj),
- 10 (el pulso dura dos ciclos de reloj) y
- 11 (el pulso de entrada está a 1 durante tres ciclos de reloj).

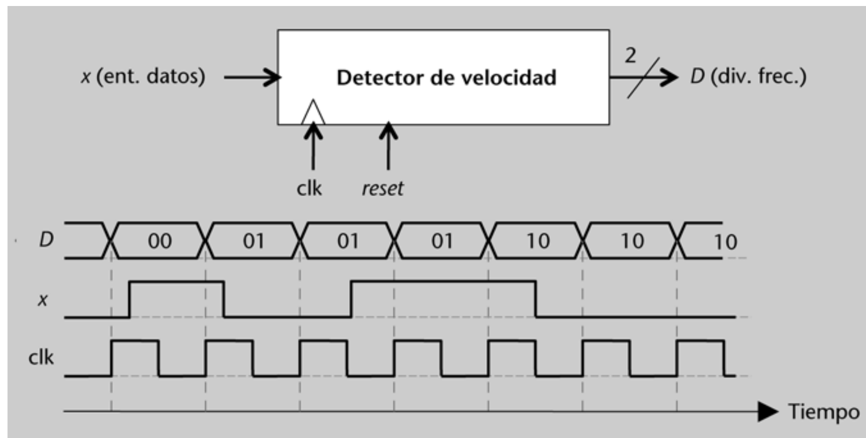
Si un pulso de entrada supera los tres ciclos de reloj, la salida también será 11.

A modo de ejemplo, en la figura 7 se ilustra el funcionamiento del detector de velocidad.

Observad que, en el segundo pulso a 1 de *x*, la salida pasa de 00 a 10 progresivamente. Es decir, cuenta el número de ciclos en el que se detecta la entrada a 1. Además, permanece en el mismo estado cada vez que finaliza una cuenta, ya que, justo después de un ciclo con la entrada a 0, el circuito mantiene la salida en la última cuenta que ha hecho.

Elaborad el diagrama de estados y la tabla de transiciones correspondiente.

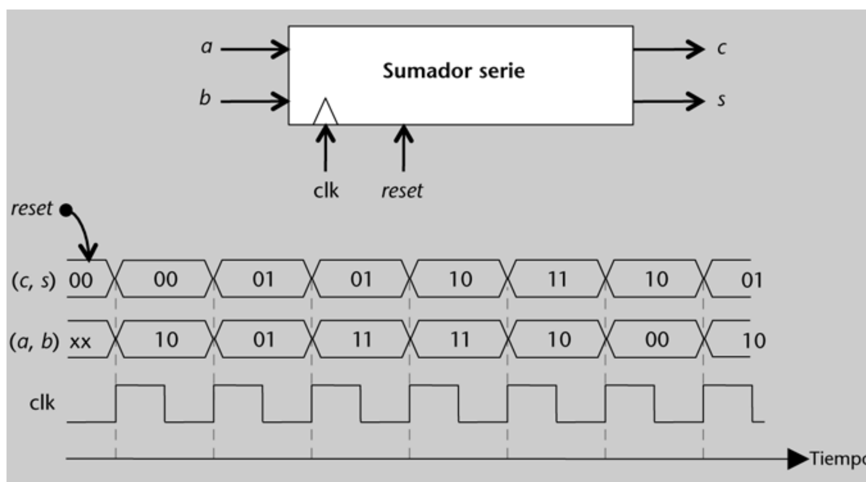
Figura 7. Esquema del detector de velocidad de transmisión




**2.** Diseñad un sumador en serie. Es un circuito secuencial con dos entradas, *a* y *b*, y dos salidas, *c* y *s*, que son, respectivamente, el acarreo (*carry*) y el resultado (suma) de la operación de suma de los bits *a* y *b* junto con el posible acarreo anterior. Inicialmente, la salida debe ser (0, 0). Es decir, el acarreo inicial es 0 y el bit menos significativo del número que se forma con la serie de salida es 0.

En la figura 8 hay una secuencia de valores de entrada y las salidas correspondientes. Fijaos en que el acarreo de salida también se tiene en cuenta en el cálculo de la suma siguiente.

Figura 8. Esquema de un sumador en serie




Primero debéis hacer el diagrama de estados y después la tabla de verdad de las funciones de transición correspondientes. Comparad el resultado de las funciones de transición con las de un sumador completo.

### 1.2. Máquinas de estados finitos extendidas

Las FSM de los controladores tratan con entradas (estados de los sistemas que controlan) y salidas (acciones) que son, de hecho, bits. Ahora bien, a menudo, los datos de entrada y los de salida son valores numéricos que, evidentemente, también se codifican en binario, pero que tienen una anchura de unos cuantos bits.

En estos casos, existe la opción de representar cada condición de entrada o cada posible cálculo de salida con un único bit. Por ejemplo, que el nivel de un depósito no supere un umbral se puede representar por una señal de un único bit, que se puede denominar “por_debajo_de_umbral”, o que el grado

de apertura de una válvula se tenga que incrementar puede verse como una señal de un bit con un nombre como "abre_más".

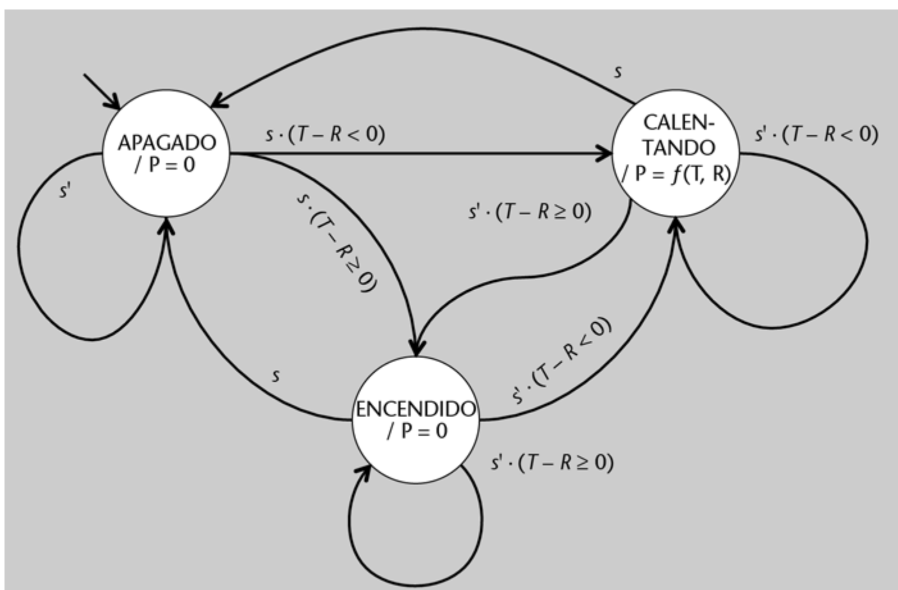
Con todo, es más conveniente incluir en los grafos de estados no solo los bits de entrada y los de salida, sino también las evaluaciones de condiciones sobre valores de entrada y los cálculos para obtener los valores de salida de una manera directa, sin tener que hacer ninguna traducción inicial a valores de uno o pocos bits. Es decir, el modelo de máquinas de estados finitos "se extiende" para incluir operaciones con datos. De aquí que se hable de **FSM extendidas** o **EFSM** (de *extended* FSM).

Para ilustrarlo, se puede pensar en un calefactor con un termostato que lo controle: en función de la diferencia entre la temperatura deseada (la referencia) y la medida (el estado del sistema que se controla) se determina la potencia del calefactor (el estado siguiente del controlador). En este caso, el controlador recibe datos de entrada que son valores numéricos y determina a qué potencia pone el radiador, un número entero entre 0 y 2, por ejemplo.

El funcionamiento del calefactor es sencillo. Cuando está apagado, la potencia de calefacción debe ser cero. Cuando se enciende, puede estar en dos estados diferentes: calentando, si la diferencia de temperatura entre la medida y la de referencia es negativa, o activo (encendido, pero no calentando) si la temperatura que se detecta es igual o superior a la que se quiere.

El grafo de estados siguiente es el correspondiente a un modelo EFSM del sistema que se ha comentado. En el grafo, *s* es una señal de entrada que actúa de conmutador (*switch*, en inglés) y que, cada vez que se activa (se pone a 1), apaga o enciende el termostato según si está encendido o apagado, respectivamente; *T* es un número binario que representa la temperatura que se mide en cada momento; *R* el valor de la referencia (la temperatura deseada), y *P* un número binario entre 0 y 2 que se calcula en función de *T* y de *R*, $f(T, R)$.

Figura 9. Grafo de estados de un termostato



En la EFSM del ejemplo se combinan señales de control de un único bit con señales de datos de más de un bit. De las últimas, hay dos de entrada, $T$ y $R$, y una de salida, $P$.

Hay que tener presente que los cálculos de las condiciones como $s \cdot (T - R < 0)$ o $s \cdot (T - R \geq 0)$ deben dar siempre un valor lógico. (En ambos casos, hay un resto que se compara con 0 y que da como resultado un valor lógico que se puede operar, haciendo el producto lógico, con $s$). Los cálculos para los valores de salida como $f(T, R)$ no tienen esta restricción, ya que pueden ser de más de un bit de salida, como es el caso de $P$, que utiliza dos para representar los valores 0, 1 y 2.

El diseño del circuito secuencial correspondiente se propondrá en la actividad número 3, una vez se haya visto un ejemplo de materialización de EFSM.

Las EFSM permiten, pues, tener en cuenta comportamientos más complejos, al asociar directamente cálculos de condiciones de entrada y de los resultados de las salidas a los arcos y nodos de los grafos correspondientes. Como se verá, el modelo de construcción de los circuitos que materializan las EFSM es muy similar al de las FSM, pero incluyen la parte de procesamiento de los datos binarios numéricos.

El proceso de diseño de uno de estos circuitos se ejemplificará con un **contador**, que es un elemento frecuente en muchos circuitos secuenciales, ya que es la base de los temporizadores, relojes, velocímetros y otros componentes útiles para los controladores.

En este caso, el contador que se diseñará es similar a los que ya se han visto en el módulo anterior, pero que cuenta hasta un número dado, $M \geq 1$. Es decir, es un contador de 0 a $M - 1$ "programable", que no empezará a contar hasta que no llegue una señal de inicio (*begin*) y que se detendrá automáticamente al llegar al valor $M - 1$. $M$ se leerá con la señal de inicio. También hay una señal de salida de un bit que indica cuándo se ha llegado al final (*end*) de la cuenta. Al arrancar el funcionamiento, la señal *end* debe estar a 1 porque el contador no está contando, lo que equivale a haber acabado una posible cuenta anterior.

Por comparación con un contador autónomo de 0 a $M - 1$, como los vistos en el módulo anterior y que se pueden construir a partir de una FSM sin entradas, lo que se propone ahora es hacer uno que lo haga dependiendo de una señal externa y, por lo tanto, que no sea autónomo.

Los contadores autónomos de 0 a $M - 1$ tienen $M$ estados y pasan de uno a otro según la función siguiente:

$$C^+ = (C + 1) \text{ MOD } M$$

Es decir, el estado siguiente ($C^+$) es el incremento del número que representa el estado ($C$) en módulo $M$.

Esta operación se puede efectuar mediante el producto lógico entre la condición de que el estado tenga el código binario de un número inferior a ($M$ – 1) y cada uno de los bits del número incrementado en 1:

$$C^+ = (C + 1) \text{ AND } (C < (M - 1))$$

A diferencia del contador autónomo, el programable, además de los estados asociados a cada uno de los $M$ pasos, debe tener estados de espera y de carga de valor límite, $M$, que es externo. Hacer un grafo de estados de un contador así no es nada práctico: un contador en el que $M$ y $C$ fueran de 8 bits tiene, como mínimo, $2^8 = 256$ estados.

Las EFSM permiten representar el comportamiento de un contador de manera más compacta porque los cálculos con $C$ pueden dejarse como operaciones con datos almacenados en una variable asociada, aparte del cálculo del estado siguiente.

> Una **variable** es un elemento del modelo de EFSM que almacena un dato que puede cambiar (o variar, de aquí su nombre) de modo similar a como se cambia de estado. De la misma manera, la materialización del circuito correspondiente necesitará un registro para cada variable, además del registro para el estado.

Las variables son análogas a los estados: pueden cambiar de valor en cada transición de la máquina. Así pues, tal como hay funciones para el cálculo de los estados siguientes, hay funciones para el cálculo de los valores siguientes de las variables. Por este motivo se utiliza la misma notación (un signo más en posición de superíndice) para indicar el valor siguiente del estado $s$ ($s^+$) y para indicar el valor siguiente de una variable $v$ ($v^+$).

Desde otra perspectiva, se puede ver una variable como un elemento interno que, por una parte, proporciona un dato de entrada (en el contador, sería $C$) y, por otra, recibe un dato de salida que será la entrada en el ciclo siguiente (en el contador, sería $C^+$). En otras palabras, el contenido actual de una variable (en el contador, $C$) forma parte de las entradas de la EFSM y, entre las salidas de la misma máquina, se encuentra la señal que transporta el contenido de la misma variable al estado siguiente (en el contador, $C^+$).

**Variable de una EFSM**

En general, cualquier variable de una EFSM puede transformarse en información de estado, generando una máquina de estados con un número de estados que puede llegar a ser tan grande como el producto del número de valores diferentes de la variable por el número de estados que tenía la EFSM original. Por ejemplo, la transformación de la EFSM en una FSM implica transformar todos los valores de la variable $C$ en estados de la máquina resultante, ya que "tiene que recordar" cuál es el valor de la cuenta actual. (En este caso, el producto es por 1, ya que la EFSM original sólo tiene un estado).

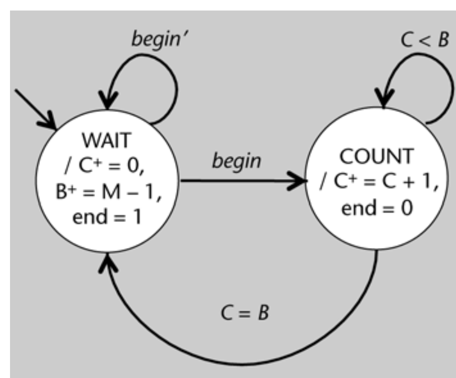
De esta manera, se puede hacer un modelo de EFSM para un contador autónomo que tenga un único estado en el que la acción consista en cargar el contenido de hacer el cálculo de $C^+$ al registro correspondiente.

Siguiendo con los modelos de EFSM para los contadores programables, hay que tener en cuenta que deben tener un estado adicional. Así pues, la EFSM de un contador programable tiene un estado más que la de uno que sea autóno-

mo, es decir, tiene dos. Uno de espera (WAIT) a que se active la señal de entrada *begin* y el necesario para hacer la cuenta (COUNT) hasta el máximo prefijado, *M*. En el primero, además de esperar a que *begin* sea 1, se actualiza el valor de la variable *B* con el valor de la entrada *M*, menos 1. Así, de iniciar una cuenta, *B* contendrá el mayor número al que se ha de llegar. En otras palabras, *B* es la variable que ayuda a recordar el máximo de la cuenta, con independencia de cualquier valor que haya en la entrada *M* en cualquier momento posterior al inicio de la cuenta.

El contador programable tiene dos salidas: la del propio valor de la cuenta, que se almacena en una variable *C*, y otra de un bit, *end*, que indica cuándo está contando o cuándo está a la espera. El grafo de estados correspondiente a este comportamiento se muestra en la figura 10.

Figura 10. Grafo de estados de un contador "programable"




Del comportamiento representado en el grafo de la figura 10 se puede ver que el contador espera a recibir un 1 por *begin* para empezar a contar. Hay que tener presente que, mientras está contando (es decir, en el estado COUNT), el valor de *begin* no importa. Después de la cuenta, siempre pasa por el estado WAIT y, por lo tanto, entre una cuenta y la siguiente debe transcurrir, como mínimo, un ciclo de reloj.

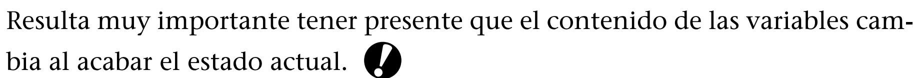
Como ya se ha visto en el ejemplo del calefactor con termostato, las entradas y las salidas no se representan en binario, sino que se hace con expresiones simbólicas, es decir, con símbolos que representan operadores y operandos. Para las transiciones se utilizan expresiones que, una vez evaluadas, deben tener un resultado lógico, como $C < B$ o *begin'*. Para los estados, se utilizan expresiones que determinan los valores de las salidas que se asocian a ellos.

En este caso, hay dos posibilidades:

**1)** Para las señales de salida, se expresa el valor que deben tener en aquel estado. Por ejemplo, al llegar al estado COUNT, *end* será 0.

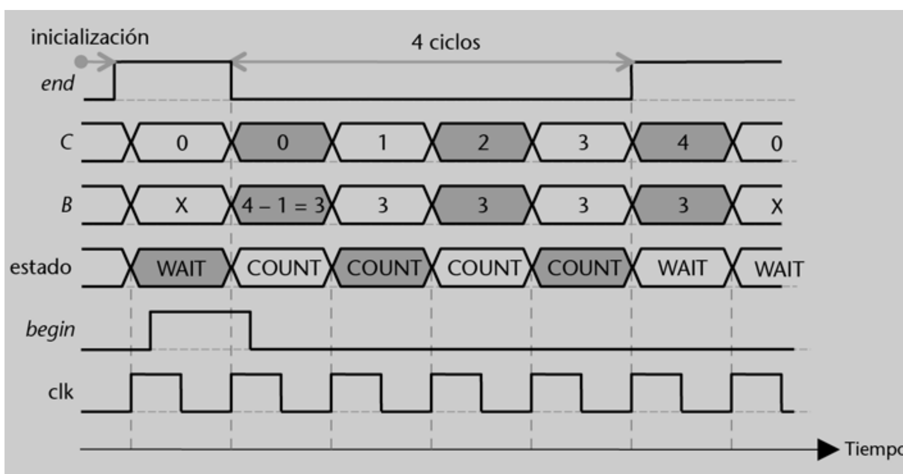
**2)** Para las variables, se expresa el valor que adquirirán **en el estado siguiente**. Por ejemplo, en el estado COUNT, la variable *C* se debe incrementar en una unidad para el estado siguiente. Para que quede claro, se pone el superíndice con el signo más: $C^+ = C + 1$.

Para que quede más claro el funcionamiento de este contador programable, se muestra a continuación un diagrama de tiempo para una cuenta hasta 4. Se trata de un caso que se inicia después de un *reset*, motivo por el cual *end* está a 1 y $C$ a 0. Justo después del *reset*, la máquina se encuentra en el estado inicial de WAIT, la variable $B$ ha adquirido un valor indeterminado $X$, $C$ continúa estando a 0 y la señal de salida *end* es 1 porque no está contando. Estando en este estado recibe la activación de *begin*, es decir, la indicación de empezar la cuenta y, junto con esta indicación, recibe el valor de 4 por la entrada $M$ (señal que no se muestra en el cronograma). Con estas condiciones, el estado siguiente será el de contar (COUNT) y el valor siguiente de las variables $C$ y $B$, será 0 y 4 – 1, respectivamente. Por lo tanto, al llegar a COUNT, $B = 3$ y $C = 0$.



Resulta muy importante tener presente que el contenido de las variables cambia al acabar el estado actual.

Figura 11. Cronograma de ejemplo para el contador




Obsérvese que el número de ciclos que transcurren entre el ciclo de reloj en el que se ha puesto *end* a 0 y el que marca el final (*end* = 1) es igual a $M$. También se puede observar que las variables toman el valor de la salida correspondiente al inicio del estado siguiente. Por este motivo, $C = 4$ en el primer WAIT después de acabar la cuenta. (Eso se podría evitar haciendo que $C^+$ se calculara con módulo $M$, de manera que, en lugar de asignársele el valor 4, se le asignara un 0).

A partir de los grafos de estados se obtienen las tablas de transiciones de estados y de salidas siguientes.

| Estado actual | Entrada | Estado futuro |
|---|---|---|
| WAIT | *begin'* | WAIT |
| WAIT | *begin* | COUNT |
| COUNT | $(C < B)$ | COUNT |
| COUNT | $(C = B)$ | WAIT |

| Estado actual | Salida |
|---|---|
| WAIT | $C^+ = 0$, $B^+ = M - 1$, *end* = 1 |
| COUNT | $C^+ = C + 1$, $B^+ = B$, *end* = 0 |

En este caso, en la tabla de salidas se han especificado los valores para las variables en todos los casos, aunque no cambien, como es el caso de la variable $B$ en COUNT.

De cara a la materialización del circuito correspondiente, hay que codificar en binario toda la información:

- La codificación de las señales de un único bit, sean de entrada o de salida, es directa. En el ejemplo, son *begin* y *end*.

- Los términos de las expresiones lógicas de las transiciones que contienen referencias a valores numéricos se transforman en señales de entrada de un único bit que representan el resultado lógico. Normalmente, son términos que contienen operadores de relación. Para el contador hay uno que determina si C < *B* y otro para *C* = *B*. A las señales de un bit que se obtienen se las referirá con los términos entre paréntesis, tal como aparecen en la tabla anterior.

- Las expresiones que indican los cálculos para obtener los valores de las señales numéricas de salida pueden ser únicas para cada uno o no. Lo más habitual es que una determinada señal de salida pueda tener distintas opciones. En el contador, *C* puede ser 0 o *C* + 1, y *B* puede ser *B* o el valor de la entrada *M*, disminuido en una unidad. Cuando hay varias expresiones que dan valores diferentes para una misma salida, se introduce un "selector", es decir, una señal de distintos bits que selecciona cuál de los resultados se debe asignar a una determinada salida. Así pues, serían necesarios un selector para *C* y otro para *B*, los dos de un bit, ya que hay dos casos posibles para cada nuevo valor de las variables *B* y *C*. La codificación de los selectores será la de la tabla siguiente:

| Operación | Selector | Efecto sobre la variable |
|---|---|---|
| $B^+ = B$ | 0 | Mantenimiento del contenido |
| $B^+ = M - 1$ | 1 | Cambio de contenido |
| $C^+ = 0$ | 0 | Almacenaje del valor constante 0 |
| $C^+ = C + 1$ | 1 | Almacenaje del resultado del incremento |

Hay que tener en cuenta que los selectores se materializan, habitualmente, como entradas de control de multiplexores de buses conectados a las salidas correspondientes. En este caso, son señales de salida ligadas a variables y, por lo tanto, a los registros pertinentes. Por este motivo, los efectos sobre las variables son equivalentes a que los registros asociados carguen los valores de las señales de salida que se seleccionen.

**Expresiones de resultado lógico**

Las expresiones de resultado lógico también se pueden identificar con señales de un único bit que, a su vez, se deben identificar con nombres significativos, como, por ejemplo, *CltB*, de "C *is less than* B", y *CeqB*, de "C *is equal to* B".

En las tablas siguientes se muestra la codificación de las entradas y de las salidas tal como se ha establecido. En cuanto a las entradas, se debe tener en cuenta que las condiciones (*C* < *B*) y (*C* = *B*) son opuestas y, por lo tanto, es suficiente, por ejemplo, con utilizar (*C* < *B*) y (*C* < *B*)', que es equivalente a (*C* = *B*). En la tabla, se muestran los valores de las dos señales. La codificación de los estados es una codificación directa, con el estado inicial WAIT con código 0. Las señales de salida son *sB* (selector de valor siguiente de la variable *B*), *sC* y *end*. El valor de *C* también es una salida del contador.

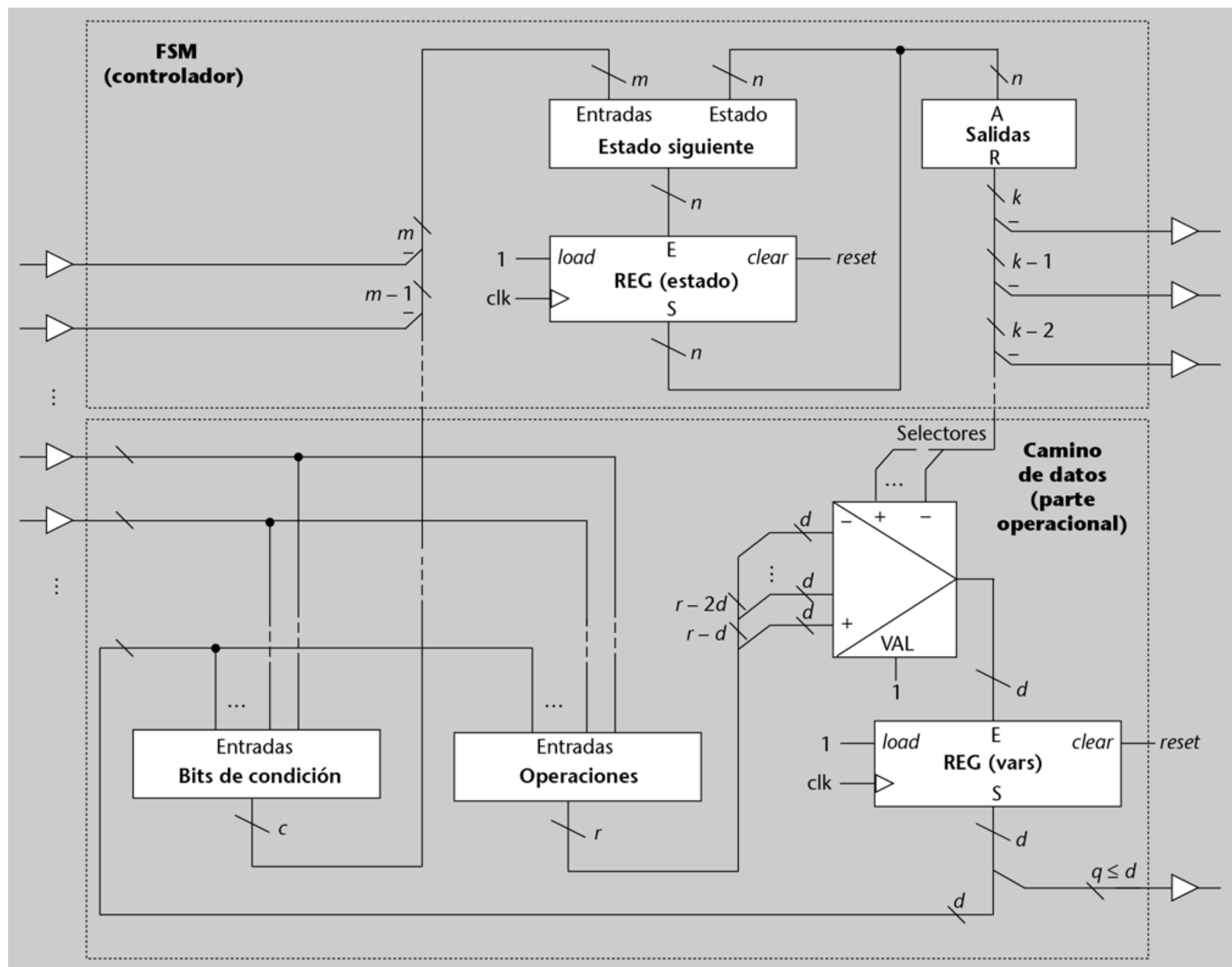
| Estado actual | | Entradas | | | Estado+ |
|---|---|---|---|---|---|
| Símbolo | *s* | *begin* | *C* < *B* | *C* = *B* | $s^+$ |
| WAIT | 0 | 0 | x | x | 0 |
| WAIT | 0 | 1 | x | x | 1 |
| COUNT | 1 | x | 0 | 1 | 0 |
| COUNT | 1 | x | 1 | 0 | 1 |

| Estado | Salidas | | |
|---|---|---|---|
| | *sB* | *sC* | *end* |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

Antes de pasar a la implementación de las funciones de las tablas de verdad anteriores, hay que ver cómo se construye una de estas EFSM. En general, la arquitectura de este tipo de circuitos secuenciales separa la parte que se ocupa de realizar las operaciones con los datos numéricos de la parte que trata con las señales de un bit y de los selectores.

En la figura 12 se puede ver la organización por módulos de un circuito secuencial para una EFSM. Se distinguen dos partes. La que aparece en el recuadro superior es la de la FSM, que recibe, como entradas, las señales de entrada de un único bit y también los resultados lógicos de las operaciones con los datos (bits de condición) y que tiene, como salidas, señales de un único bit y

Figura 12. Arquitectura de una EFSM



también selectores para los resultados que se deben almacenar en los registros internos. La segunda parte es la que realiza las operaciones con datos o **camino de datos**. Esta parte recibe, como entradas, todas las señales de entrada de datos de la EFSM y los selectores y, como salidas, proporciona tanto los bits de condición para la FSM de la parte controladora como los datos numéricos que corresponden a los registros ligados a las variables internas de la EFSM. Adicionalmente, las salidas de parte de estos registros pueden ser, también, salidas de la EFSM.

En resumen, pues, la EFSM tiene dos tipos de entradas (las señales de control de un bit y las señales de datos de múltiples bits) y dos tipos de salidas (señales de un único bit, que son de control para elementos exteriores, y señales de datos resultantes de alguna operación con otros datos, internos o externos). El modelo de construcción que se ha mostrado las organiza de manera que los bits de control se procesan con una FSM y los datos con una unidad operacional diferenciada. La FSM recibe, además, señales de un bit (condiciones) de la parte operacional y le proporciona bits de selección de operación para las salidas. Esta parte aprovecha estos bits para generar las salidas correspondientes, entre las que se encuentran, también, el valor siguiente de las variables. Al mismo tiempo, parte del contenido de los registros puede ser utilizado, también, como salidas de la EFSM.

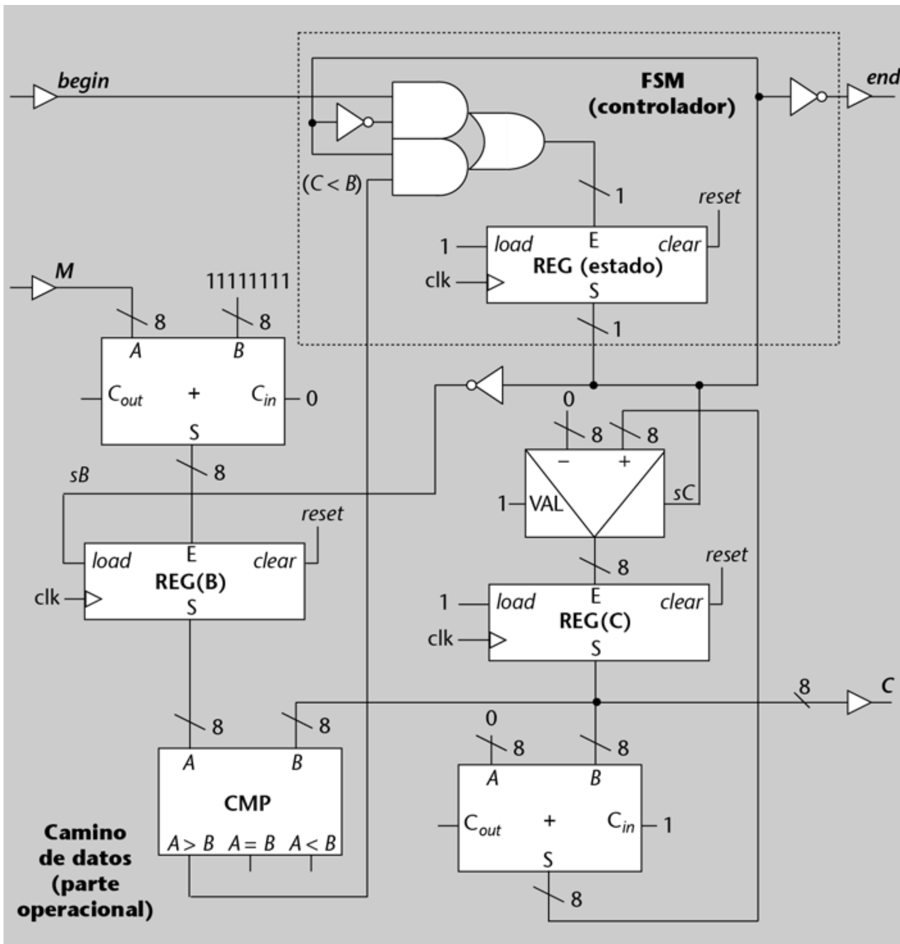
Esta arquitectura, que separa el circuito secuencial en dos unidades: la de control y la de procesamiento, se denomina **arquitectura de máquina de estados con camino de datos** o **FSMD**, que es el acrónimo del inglés *finite-state machine with datapath*.

Para la construcción del contador se debe seguir el mismo modelo. En la figura 13 se puede ver el circuito completo. La parte superior se corresponde con la unidad de control, que calcula la función de excitación, que es $s^+ = s' \cdot begin + s \cdot (C < B)$, y las de salida (*end*, hacia el exterior, y *sB* y *sC*, que son selectores), que se obtienen directamente del estado o del estado complementado. Se ha mantenido el registro de estado, como en el esquema anterior, pero es un registro de un único bit que se puede implementar con uno biestable. Se ha hecho una pequeña optimización: el bus de entrada para el registro *B* no lo proporciona la salida de un multiplexor controlado por *sB*, sino que es directamente el valor $M - 1$, que se carga o no, según *sB*, que está conectado a la entrada *load* del registro *B*. (En general, esto siempre se podrá hacer con las variables que tengan, por una parte, el cambio de valor y, por la otra, el mantenimiento del contenido).

En el circuito se puede observar que las variables de las EFSM proporcionan valores de entrada y almacenan valores de salida: el registro *C* proporciona el valor de la cuenta en el periodo de reloj en curso y, después, almacenará el valor de salida que se calcule para $C^+$, que depende del selector *sC* y, en última instancia, del estado actual. En consecuencia, el valor de *C* no se actualiza hasta que no se pasa al estado siguiente.

Hay que tener en cuenta que, en las EFSM, el estado completo queda formado por el estado de la FSM de control y el estado del camino de datos, que se define por el contenido de los registros de datos internos. Como la funcionalidad de las EFSM queda fijada por la parte controladora, es habitual hacer la equivalencia entre sus estados, dejando aparte los de los caminos de datos. Así, cuando se habla de estado de una EFSM se hace referencia al estado de su unidad de control.

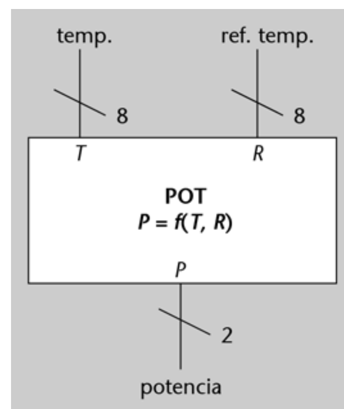
Figura 13. Circuito secuencial correspondiente al contador del ejemplo




## Actividades

**3.** Implementad el circuito de la EFSM del termostato. Para ello, se dispone de un módulo, POT, que calcula $f(T, R)$.

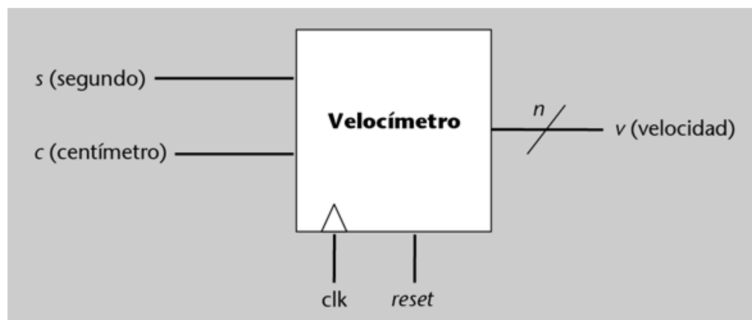
Figura 14. Esquema de entradas y salidas del módulo POT



**4.** En muchos casos, de los contadores solo interesa la señal de final de cuenta. En este sentido, no es necesario que la cuenta se haga de 0 a $M - 1$, tomando la notación de lo que se ha visto, sino que también se puede hacer al revés. De esta manera, es suficiente con un único registro cuyo valor vaya decreciendo hasta 0. Modificad la EFSM del contador del ejemplo para que se comporte de esta manera y diseñad el circuito resultante.

**5.** Un velocímetro consiste en un mecanismo que cuenta unidades de espacio recorrido por unidades de tiempo. Por ejemplo, puede contar centímetros por segundo. Se trata de diseñar el grafo de estados de un controlador de un velocímetro que tome como entradas una señal *s*, que se pone a 1 durante un ciclo de reloj en cada segundo, y una señal *c*, que es 1 en el periodo de reloj en el que se ha recorrido un centímetro, y, como salida, el valor de la velocidad en cm/s, *v*, que se debe actualizar a cada segundo. Hay que tener presente que *s* y *c* pueden pasar en el mismo instante. El valor de la velocidad se debe mantener durante todo un segundo. Al arrancar, durante el primer segundo deberá ser 0.

Figura 15. Esquema de entradas y salidas del módulo del velocímetro




## 1.3. Máquinas de estados-programa

Las máquinas de estados extendidas permiten integrar condiciones y operaciones en los arcos y nodos de los grafos correspondientes. No obstante, muchas operaciones no son simples e implican la ejecución de un conjunto de operaciones elementales en secuencia, es decir, son **programas**. Por ejemplo, una acción que dependa de una media calculada a partir de distintas entradas requiere una serie de sumas previas.

En este sentido, aunque se puede mantener esta secuencia de operaciones de manera explícita en el grafo de estados de la máquina correspondiente, resulta mucho más conveniente que los programas queden asociados a un único nodo. De esta manera, cada nodo representa un estado con un posible programa asociado. Por este motivo, a las máquinas correspondientes se las denomina **máquinas de estados programa** o **PSM** (de las siglas en inglés de *program-state machines*). Las PSM se pueden ver como representaciones más compactas de las EFSM.

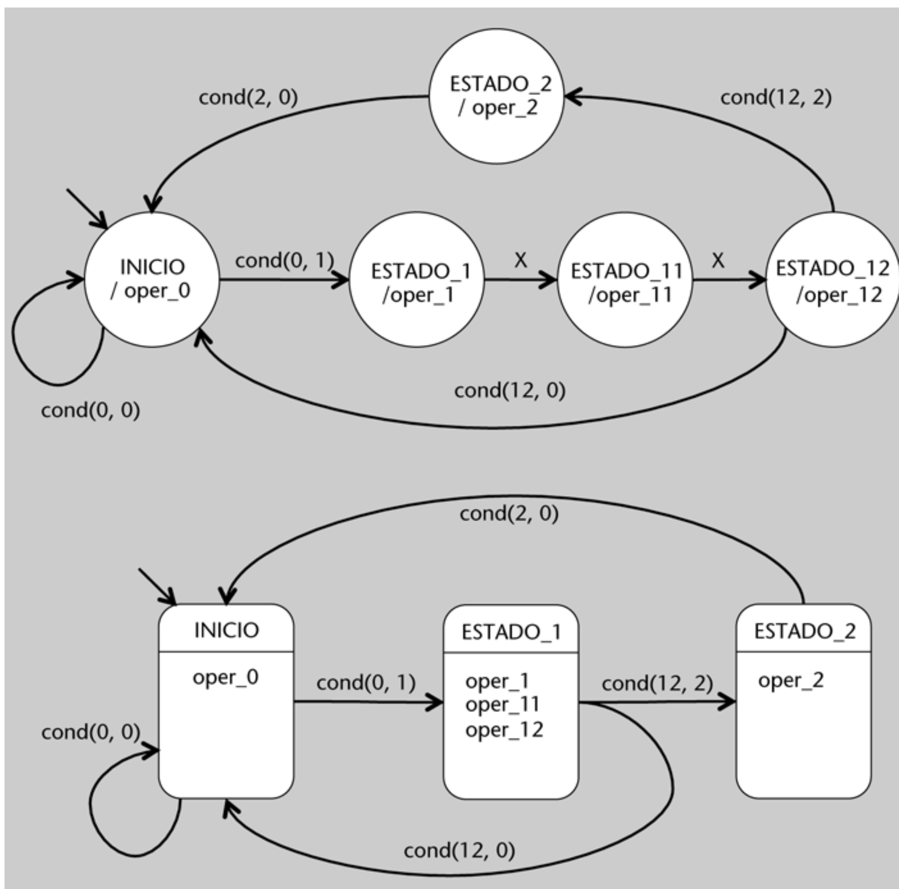
Los programas de cada estado son, de hecho, un conjunto de acciones que hay que ejecutar en secuencia. (En el modelo general de PSM, los programas no tienen ninguna restricción, pero en este texto se supondrá que son series de acciones que se llevan a cabo una tras otra, sin saltos).

Los estados a los que están vinculados pueden mantenerse durante la ejecución del programa o cambiar según las entradas correspondientes. De hecho, en el primer caso se dice que las posibles transiciones solo se efectúan una vez

acabada la ejecución del programa (*transition on completion* o TOC, en inglés) y, en el segundo caso, que se llevan a cabo de manera inmediata al iniciar la ejecución (*transition immediately* o TI, en inglés). En este texto solo se tratará de las PSM con TOC, ya que el modelo con TI queda fuera del alcance de esta asignatura.

En el ejemplo de la figura 16, hay una EFSM con una serie de nodos ligados a unas acciones que se llevan a cabo en secuencia siempre que se llega a ellos por ESTADO_1. Hay que tener en cuenta que los arcos marcados con *X* se corresponden con transiciones incondicionales, es decir, en las que no se tiene en cuenta ningún bit de condición o entrada de control.

Figura 16. Ejemplo de una EFSM (arriba) y de una PSM (abajo) equivalentes




En el modelo de PSM con TOC, estas secuencias de estados se pueden integrar en un único estado-programa (ESTADO_1). Cada estado-programa tiene asociada la acción de ejecutar, en secuencia, las acciones del programa correspondiente. En el caso del ESTADO_1, se haría primero *oper*_1, después *oper*_11 y, finalmente, *oper*_12. Hay que tener presente que cada una de estas acciones puede implicar la realización de distintas acciones subordinadas en paralelo, igual a como se llevan a cabo en los nodos de los estados de una EFSM.

Siguiendo con el ejemplo de la media de los valores de distintas entradas, las acciones en secuencia se corresponderían a la suma de un valor diferente en cada paso y, finalmente, a la división por el número de valores que se han su-

mado. Cada una de estas acciones se haría en paralelo al cálculo de posibles salidas de la EFSM. Por ejemplo, se puede suponer que la salida $y$ se mantiene en el último valor que había tomado y que, en el último momento, se pone a 1 o a 0 según si la media supera o no un determinado umbral prefijado, $T$. Este tipo de estado-programa, para cuatro valores de entrada concretos ($V_1$, $V_2$, $V_3$ y $V_4$), podría tener un programa asociado como el que se presenta a continuación:

$$\{S^+ = V_1,\ y = M \geq T\};$$
$$\{S^+ = S + V_2,\ y = M \geq T\};$$
$$\{S^+ = S + V_3,\ y = M \geq T\};$$
$$\{S^+ = S + V_4,\ y = M \geq T\};$$
$$\{M^+ = S/4,\ y = S/4 \geq T\}$$

Este programa utiliza, además, dos variables para contener la suma ($S$) y la media ($M$) de los valores de entrada. Hay que tener presente que:

- las llaves indican grupos de cálculos que se hacen en un único paso, es decir, incluyen todas las acciones que se realizan en un mismo periodo de tiempo en paralelo;

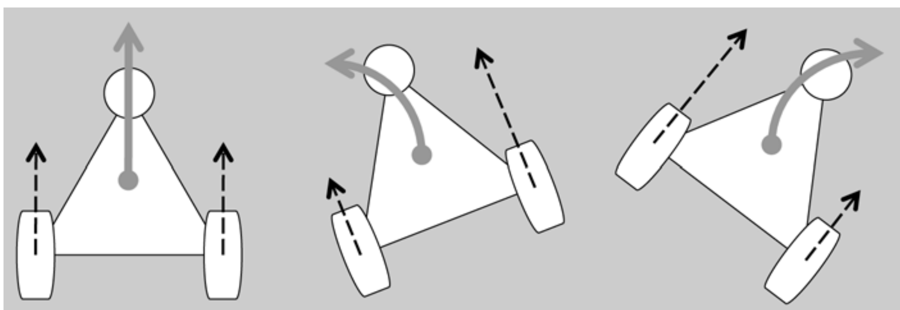
- las comas separan las diferentes acciones o cálculos, y

- los puntos y comas separan dos grupos de acciones que se deben hacer en secuencia o, lo que sería lo mismo, que en una EFSM se corresponderían con estados diferentes.

Para ver las ventajas de las PSM, se trabajará con un ejemplo realista: el diseño de un controlador de velocidad de un servomotor para un robot.

Habitualmente, los robots sencillos están dotados de un par de servomotores que están conectados a las correspondientes ruedas y se mueven de manera similar a la de un tanque o cualquier otro vehículo con orugas: si se hacen girar las dos ruedas a la misma velocidad y en el mismo sentido, el vehículo se mueve adelante o atrás; si las ruedas giran a diferente velocidad o en diferente sentido, el vehículo describirá una curva según el caso:

Figura 17. Movimiento por "par motriz diferencial"



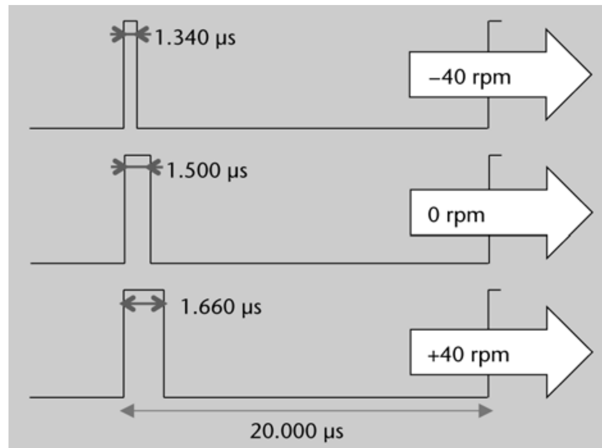
**¿Cómo funcionan los servomotores?**

Los **servomotores** son unos motores que tienen una señal de control que, en función del ancho de los pulsos a 1 que tenga, hacen girar el rotor hasta una cierta posición. Conve-

nientemente adaptados, eso se transforma en que el ancho de los pulsos determina a qué velocidad y en qué sentido han de girar. Estos pulsos deben tener una cierta periodicidad para mantenerlos funcionando. De hecho, la señal de control describe una forma de onda que "transporta información" en el ancho de estos pulsos periódicos y, por eso mismo, se dice que es una señal con modulación de amplitud de pulso o PWM (de *pulse-width modulation*, en inglés). La modulación hace referencia, precisamente, al cambio que sufre el ancho del pulso (en este caso) según la información que se transmite.

Para los servomotores, es habitual que los anchos de los pulsos se sitúen entre 1.340 y 1.660 microsegundos (μs), con una periodicidad de un pulso cada 20.000 μs, tal como se muestra en la figura 18. En ausencia de pulso, el motor permanece inmóvil, igual que con pulsos de 1.500 μs, que es el valor medio. Los pulsos de anchos superiores hacen que el rotor se mueva en un sentido y los de inferiores, en el sentido contrario.

Figura 18. Ejemplo de control de un servomotor por anchura de pulso



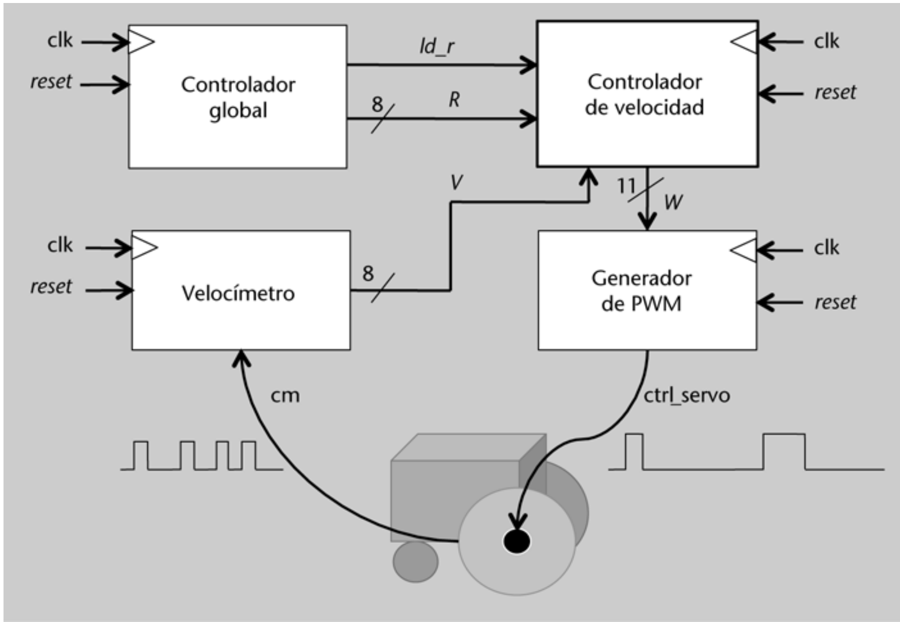

Desgraciadamente, la relación entre el ancho de pulso y la velocidad en revoluciones por minuto (rpm) no es lineal (podéis ver la figura 20) y puede cambiar a lo largo del tiempo.

El controlador de velocidad de una de las ruedas de un vehículo que se mueva con par motriz diferencial es una parte fundamental del control general del robot correspondiente. En el caso del ejemplo, será uno de los módulos del robot, que se relaciona con otros módulos del controlador completo, tal como se muestra en la figura 19.

El módulo del controlador de velocidad se ocupa de indicar cuál es el ancho de pulso que se debe generar para el servomotor correspondiente, en función tanto de lo que le indique el controlador global, como de la velocidad de giro que capte a través de un velocímetro.

Las señales que relacionan los distintos módulos se detallan a continuación. La explicación siempre es en referencia al controlador de velocidad. La señal de entrada *ld_r* se activa cuando hay una nueva velocidad de referencia *R*. La velocidad de referencia es un valor en el rango [–40, +40] y precisión de ½ rpm, por lo que se representa en formato de coma fija de 7 bits para la parte entera y 1 bit para la parte fraccionaria y en complemento a 2. La entrada *V*, que viene del velocímetro, indica la velocidad de giro de la rueda, que se mide con precisión de ½ rpm, en el mismo formato que *R*. Finalmente, la salida *W* indica el ancho del pulso para el generador de la señal modulada por ancho de pulso. El ancho de pulso es un valor expresado en microsegundos, que se sitúa en el rango [1.340, 1.660] y, por lo tanto, necesita 11 bits si se expresa en formato de número binario natural.

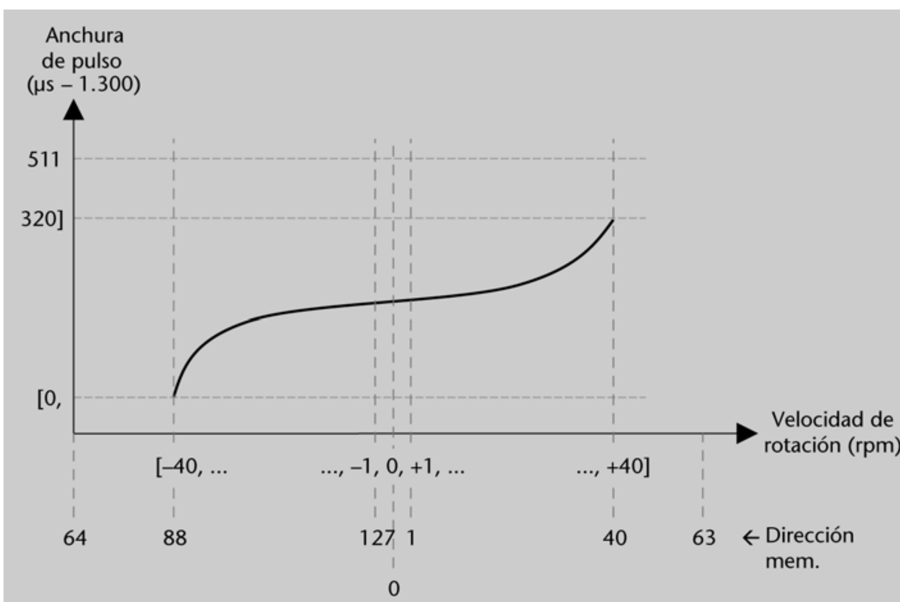
Figura 19. Esquema de los módulos de un controlador de un robot




La relación no lineal entre velocidad y ancho de pulso se almacena en una memoria ROM del controlador de velocidad. En esta memoria se almacenarán los puntos de la función que relaciona el ancho de pulso con la velocidad de rotación, tal como se muestra en la figura 20. Así, dada una velocidad tomada como dirección de la memoria, se obtiene el ancho de pulso correspondiente, tomado del contenido de la posición correspondiente.

Para minimizar las dimensiones de la ROM y sabiendo que los anchos en μs están siempre en el rango [1.340, 1.660], cada posición de memoria solo almacenará el exceso sobre 1.340 y, consiguientemente, será un valor entre 0 y 320. Por lo tanto, las palabras de memoria serán de 9 bits.

Figura 20. Representación de la función no lineal del ancho de pulso respecto a la velocidad de rotación



Para reducir el número de posiciones, hay que disminuir el número de puntos que se guardan de la función. Así, en lugar de guardar un valor de ancho de pulso para cada valor de velocidad diferente, se puede guardar uno cada dos, e interpolar los anchos por las velocidades que queden en medio. En el caso que se trata, eso coincide con almacenar los valores de los anchos de pulso para las velocidades enteras (es decir, con el bit fraccionario a 0) y, si es el caso, interpolar el valor del ancho de pulso para los valores de velocidad con el bit de ½ rpm a 1. De esta manera, las velocidades que se toman para determinar las posiciones de la memoria ROM son valores en el rango [–40, +40], que se pueden codificar en binario con 7 bits.

En resumen, la memoria ROM será de $2^7 \times 9 = 128 \times 9$ bits. Las direcciones se obtienen del valor de la velocidad de manera directa. Es decir, se tratan los 7 bits de los números como si fueran números naturales. Esto hace que los valores positivos se encuentren en las direcciones más pequeñas y los negativos, después. De hecho, si se observa la figura 20, se puede comprobar que las velocidades positivas se guardan de las posiciones 0 a 40 y las negativas, de la posición 88 (que es igual a 128 – 40) a la posición 127 (= 128 – 1). Las posiciones de memoria entre la 41 y la 87 no se utilizan.

Con la información de las entradas y de cómo se obtiene la salida ya se puede pasar a diseñar el modelo de comportamiento del controlador de velocidad, en este caso, con una PSM. Se supone que la máquina se pone en marcha a la llegada de *ld_r* y solo vuelve al estado inicial con un *reset*.

En el estado inicial, INICIO, se espera a que se active *ld_r*. Cuando se activa *ld_r* pasa al estado de NUEVA_R, donde empieza la ejecución de un programa. Como se trata de un modelo de PSM de tipo TOC, es decir, que solo efectúa transiciones una vez que se ha completado el programa asociado, no se cambia de estado hasta que no acabe la ejecución. Lo primero que hace este programa es cargar la nueva referencia *R* a dos variables, *U* y *T*; la primera sirve para almacenar *R* y la segunda para contener la velocidad de trabajo, que, inicialmente, debe ser *R*. Después convierte este valor a un valor de ancho de pulso aprovechando la función de transformación que hay almacenada en la memoria ROM.

La conversión de velocidad a ancho de pulso se lleva a cabo con una interpolación de dos puntos de la función. El valor de la velocidad se representa en números de 8 bits, uno de los cuales es fraccionario, y en Ca2. Pero la memoria solo contiene el valor de la función de transformación para puntos enteros de 7 bits. Así pues, si el bit fraccionario es 1, se considera que el valor de la función será la media de los valores anterior y posterior:

$$A = (\mathrm{M}[T_7T_6T_5T_4T_3T_2T_1] + \mathrm{M}[T_7T_6T_5T_4T_3T_2T_1 + 1])/2$$

donde $A$ es un valor entre 0 y 320 interpolado entre las posiciones de memoria en las que se encuentra el punto $T$. Los 7 bits más significativos, $T_7T_6T_5T_4T_3T_2T_1$,

constituyen la dirección de memoria donde se encuentra el valor de la función. (La división por dos consiste en desplazar el resultado un bit a la derecha.)

Si el bit fraccionario ($T_0$) es 0, el valor resultante es directamente el del primer acceso a memoria. Por simplicidad, en el programa siempre se realiza la interpolación pero, en lugar de sumar 1 a la segunda dirección, se le suma $T_0$. De esta manera, si es 1 se hace una interpolación y si no, se suma dos veces el mismo valor y se divide por 2, lo que lo deja igual. Como los valores de la memoria están sesgados respecto a 1.340, se les tiene que sumar esta cantidad para obtener el ancho de pulso en μs.

La última operación del programa del estado NUEVA_R es la activación de un temporizador para esperar a que la acción de control tenga efecto. Para hacerlo, es suficiente con poner a 1 la señal de *begin* correspondiente y esperar un 1 por la salida *end* del temporizador, que debe estar preparado para que el tiempo que transcurra entre la activación y la finalización del periodo sea suficiente como para que el motor actúe.

**Temporizador**

Un **temporizador** es un contador como los que se han visto anteriormente, con una entrada $M$ para indicarle cuántos ciclos de reloj debe estar contando, una entrada (*begin*) para iniciar la cuenta y una salida (*end*) que se pone a 1 cuando ha llegado al final de la cuenta. Si se sabe el periodo del reloj, $M$ determina el tiempo que ha de transcurrir hasta que no active la señal de finalización.

Este temporizador es un submódulo (es decir, un módulo incluido en uno mayor) que forma parte del controlador de velocidad y, por lo tanto, las señales *begin* y *end* son internas y no visibles en el exterior. El resto de la discusión se centrará en el submódulo principal, que es el que se modelará con una PSM.

La espera se efectúa en el estado MIDIENDO. De acuerdo con el diagrama de módulos de la figura 19, en esta espera el velocímetro tendrá tiempo de actualizar los cálculos de la velocidad. Al recibir la activación de la señal *end*, que indica que el temporizador ha llegado al final de la cuenta, se pasa al estado MOVIENDO.

El estado MOVIENDO es, de hecho, el corazón del controlador: calcula el error entre la velocidad de trabajo, $T$, y la velocidad medida, $V$, y lo suma a la dirección de la de referencia para corregir el desvío entre la función almacenada y la realidad observada. Si no hay error, el valor con el que actualizará la variable de salida $W$ será el mismo que ya tenía. Hay que tener en cuenta que el estado de carga de las baterías y las condiciones en las que se encuentran las ruedas asociadas tienen mucha influencia sobre los servomotores, y que esta corrección es muy conveniente.

El programa asociado es igual al de NUEVA_R, excepto que la velocidad que se convierte en ancho de pulso de salida se corrige con el error medido. Es decir, la velocidad a la que se ha de mover el motor, $T$, es la de referencia, $U$, más el error calculado, que es la diferencia entre la velocidad deseada anterior, $T$, y la que se ha medido en el momento actual, $V$:

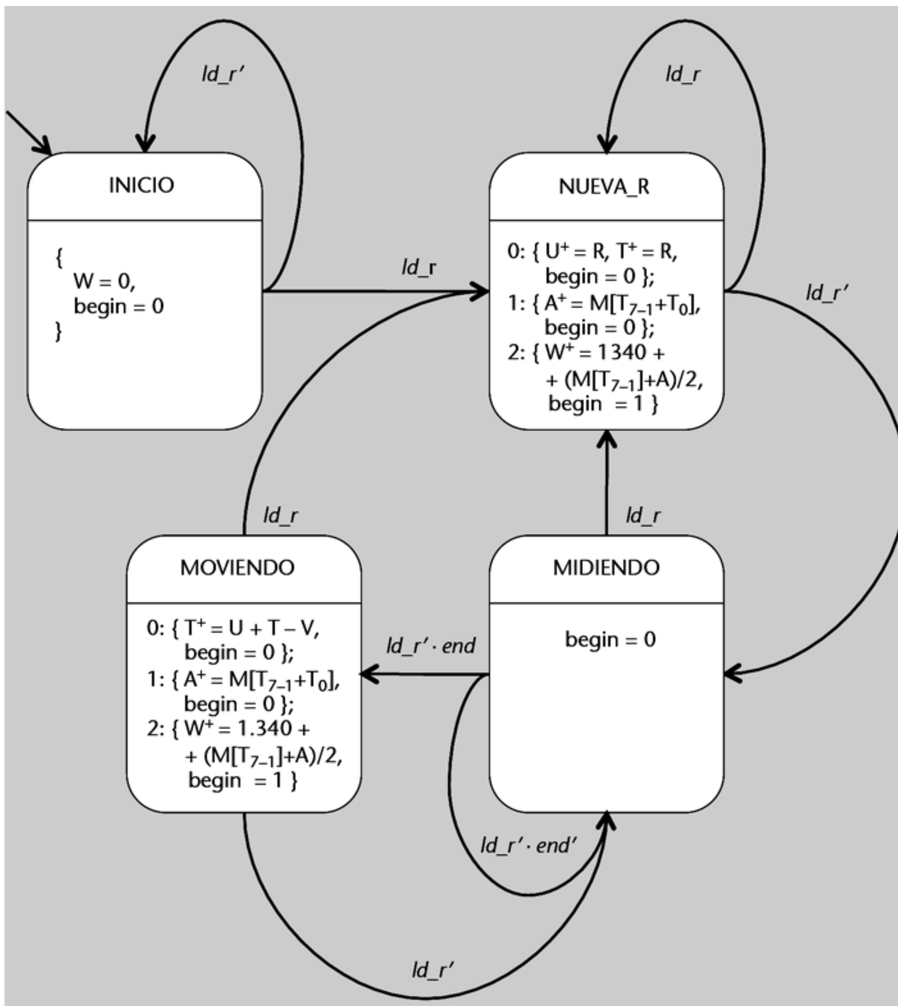
$$T^{+} = U + (T - V)$$

Por ejemplo, si la velocidad de referencia es de $U$ = 15 cm/s, la velocidad de trabajo será, inicialmente, $T$ = 15 cm/s. Si llega un momento en el que la velocidad medida es de $V$ = 12 cm/s, significa que el ancho de pulso que se tiene para los 15 cm/s no es suficiente y que hay que aumentarla un poco. En este caso se aumentará hasta el de la velocidad de $T^{+}$ = 18 cm/s. Si el ancho resultara demasiado grande y en la medida siguiente se midiera una de 16 cm/s, el cálculo anterior haría que la próxima velocidad de trabajo fuera $T^{+}$ = 15 + (18 – 16) = 17 cm/s.

En este caso, los programas de los estados MOVIENDO y NUEVA_R trabajan con las mismas variables. Como los programas ejecutan sus instrucciones en secuencia, las variables actualizan su valor después de una acción y lo tienen disponible para la siguiente. Así, el cálculo de $A^+$ se lleva a cabo con el valor calculado para $T$ en la instrucción previa, es decir, con el valor de $T^+$.

En resumen, la máquina, una vez cargada una nueva referencia en el estado-programa NUEVA_R, irá alternando entre los estados MOVIENDO y MIDIENDO mientras no se le indique cargar una nueva referencia o se le haga un *reset*. El grafo del PSM correspondiente se puede ver en la figura 21. En el diagrama, las acciones que se hacen en un mismo periodo de reloj se han agrupado con claves y los distintos pasos de cada programa, se han enumerado convenientemente.

Figura 21. Diagrama de estados del control adaptativo de velocidad



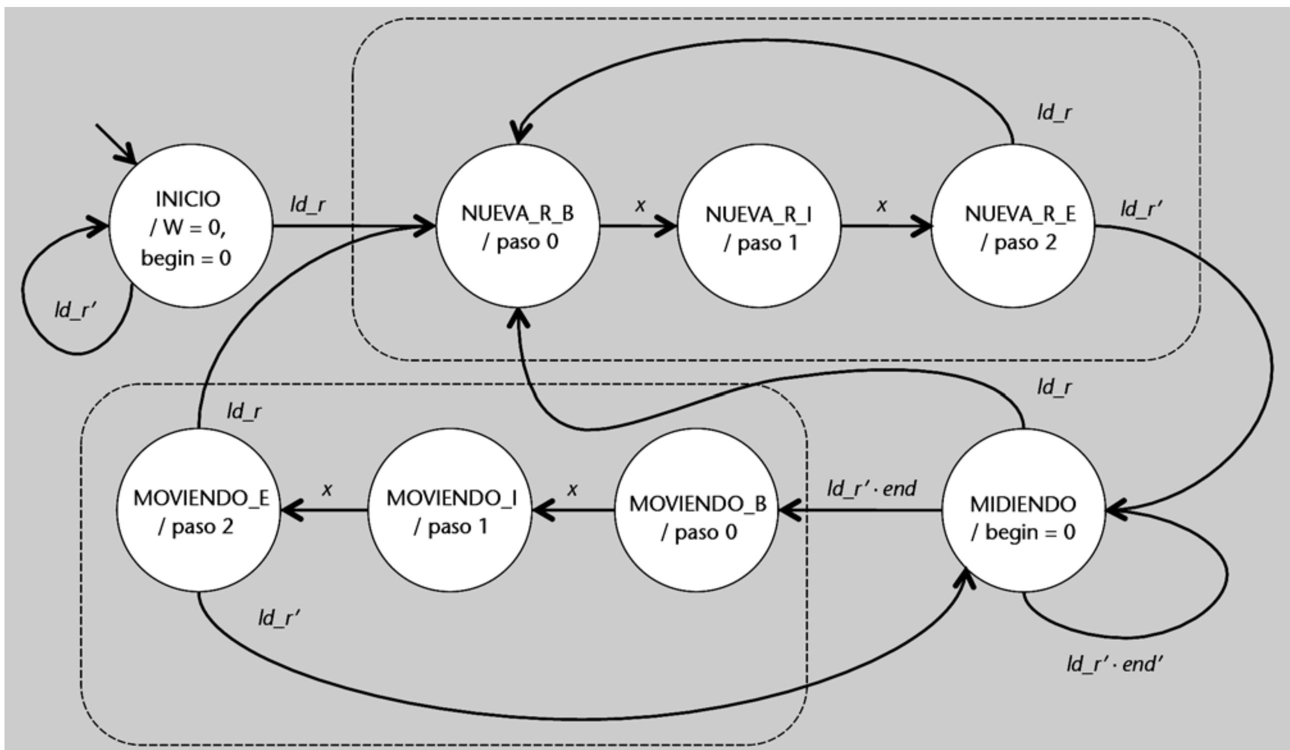

La construcción de las PSM se puede realizar de dos maneras:

**1)** por expansión de los estados-programa en secuencias de estados, de manera contraria a la mostrada en la figura 16, y

**2)** por creación de EFSM para cada estado-programa, formando un conjunto de EFSM interrelacionados.

En el primer caso, el número de estados que se debe tener en cuenta de cara a la materialización puede ser muy grande, tal como se puede comprobar en la figura 22, en la que se pasa de cuatro a ocho estados. En el grafo correspondiente, los estados-programa del caso de ejemplo NUEVA_R y MOVIENDO se han dividido en tantos subestados como pasos tienen los programas. En este caso particular, se les ha dado nombres acabados en "_B", de principio o *begin*; "_I", de estado intermedio, y "_E", de fin o *end*. Por simplicidad, las acciones asociadas a cada paso de los programas se han sustituido por una referencia al número de paso que corresponde.

Figura 22. EFSM de la PSM del controlador de velocidad



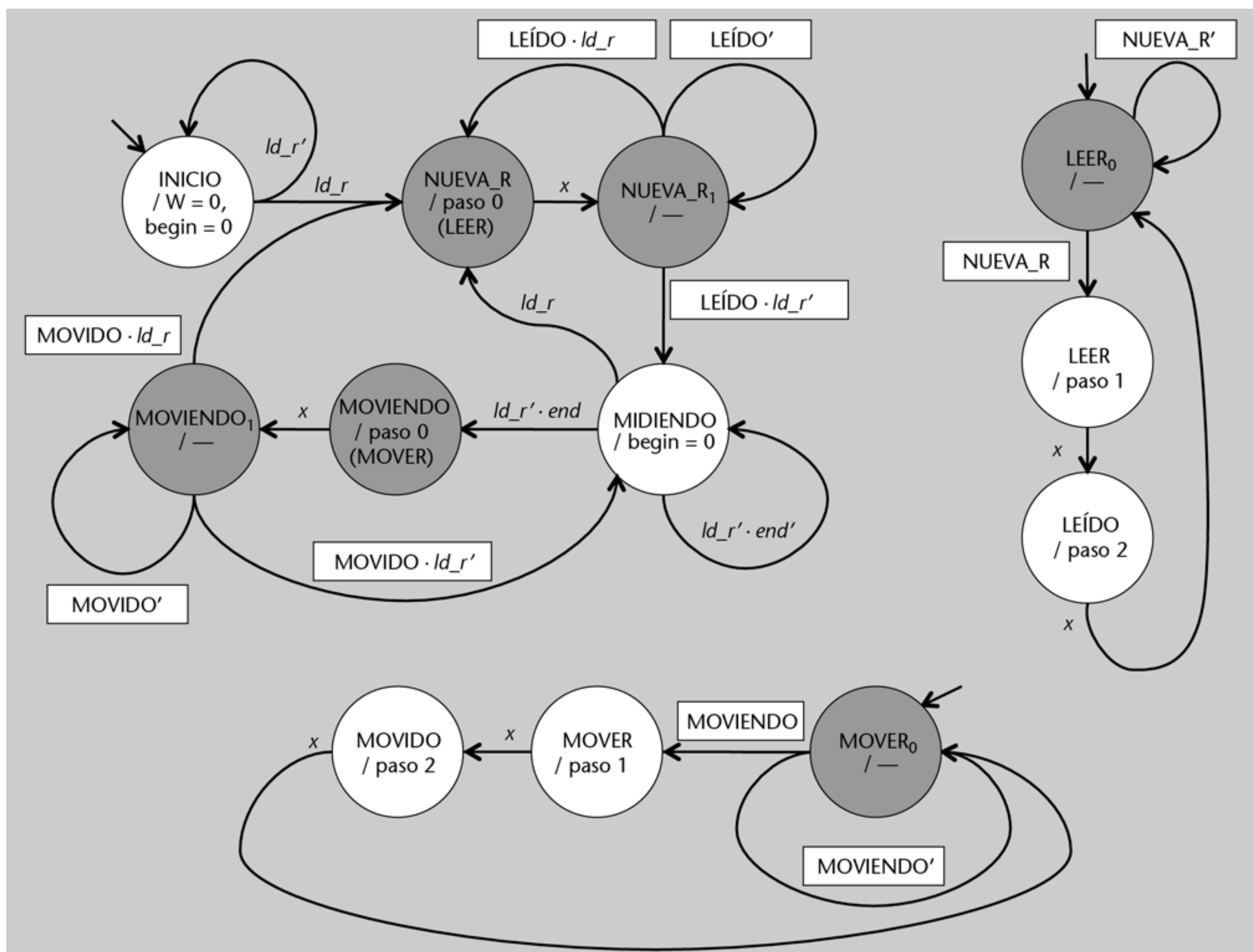

En el segundo caso, el diseño de las EFSM es mucho más simple, a cambio, como se verá, de tener que añadir estados adicionales y señales para coordinarlas.

En el ejemplo de la figura 21, la EFSM global es la formada por los estados INICIO, NUEVA_R, MIDIENDO y MOVIENDO, y hay dos EFSM más para los programas de los estados NUEVA_R y MOVIENDO. Tal como se puede observar en la figura 23, estas dos últimas EFSM están formadas por los estados $LEER_0$, LEER y LEÍDO, y por $MOVER_0$, MOVER y MOVIDO, respectivamente. También se puede observar que los estados NUEVA_R y MOVIENDO se han desdoblado en dos, añadiendo los estados $NUEVA_R_1$ y $MOVIENDO_1$ a la EFSM global.

Los estados adicionales sirven para efectuar esperas. Así pues, cuando la EFSM global llega al estado NUEVA_R o MOVIENDO, las EFSM correspondientes tienen que pasar de un estado de espera ($LEER_0$ o $MOVER_0$) a los que llevan a la secuencia de estados de los programas que tienen asociados. Al mismo tiempo, la EFSM global tiene que pasar a un estado de espera ($NUEVA_R_1$ o $MOVIENDO_1$), del que tiene que salir en el momento en que la EFSM del programa que se ha activado retorne al estado de espera propio, lo que indica que se ha finalizado la ejecución.

Para ver este comportamiento de una manera un poco más clara, se puede suponer que la EFSM principal, el global, está en el estado de INICIO y *ld_r* se pone a 1, con lo que el estado siguiente será NUEVA_R. En este estado-programa hay que llevar a cabo las acciones de los pasos 0, 1 y 2 del programa correspondiente antes de comprobar si se ha de cargar una nueva referencia o se tiene que pasar a controlar el movimiento. Por ello, en NUEVA_R se ejecuta el primer paso del programa y se activa la EFSM secundaria correspondiente a la derecha de la figura 23. En esta EFSM, el estado inicial $LEER_0$ solo se abandona cuando la EFSM principal está en el estado NUEVA_R. Entonces, se pasa de un estado al siguiente para llevar a término todos los pasos del programa hasta acabar, momento en el que retorna a $LEER_0$. En todo este tiempo, la EFSM principal se encuentra en el estado $NUEVA_R_1$, del que sale en el momento en que la EFSM secundaria ha llegado al último estado de la secuencia, LEÍDO. Cabe señalar que los arcos de salida de $NUEVA_R_1$ son los mismos que los del estado-programa NUEVA_R original, pero con las condiciones de salida multiplicadas por el hecho de que la EFSM secundaria esté en el estado LEÍDO.

Figura 23. Vista de la PSM del controlador de velocidad como conjunto de EFSM



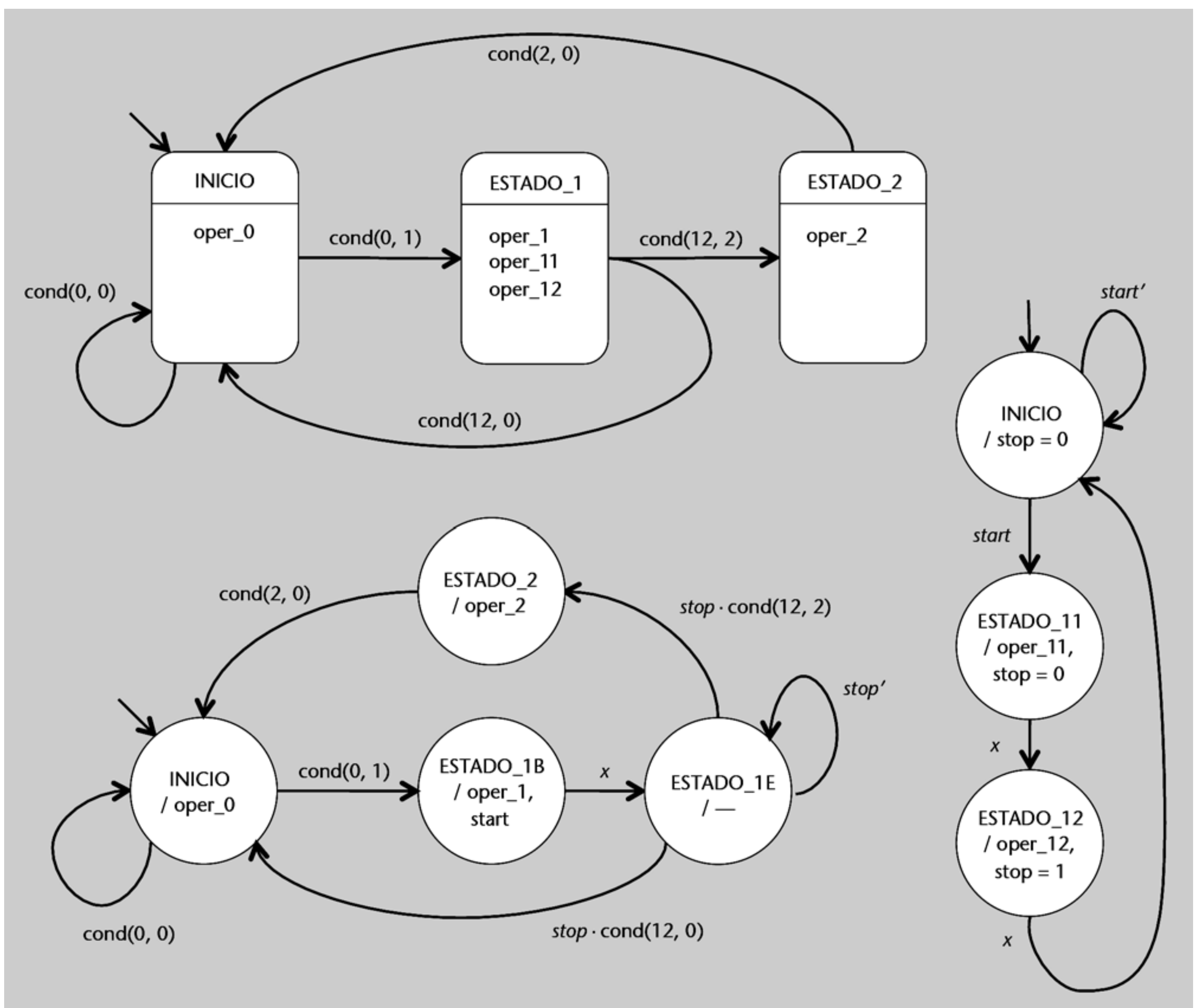

Como se ha visto, las relaciones entre las EFSM se establecen por medio de estados de espera y de referencias de estados externos a las entradas. Por ejemplo, que la EFSM de la parte inferior de la figura 23 pase de $MOVER_0$ a MOVER depende de una entrada que indique si la EFSM principal está o no en el estado MOVIENDO.

Además de la añadidura de los estados de espera, es necesario que la EFSM principal tenga una señal de salida y una de entrada para cada EFSM secundaria, de manera que pueda llevar a cabo el programa correspondiente y detectar cuándo se ha acabado la ejecución. Hay que tener presente que, desde la perspectiva de las EFSM secundarias, las señales son las mismas pero con sentido diferente: las salidas de la EFSM principal son las entradas y al revés.

Siguiendo con el ejemplo anterior, las señales de salida de la EFSM principal se podrían denominar *iniLeer* e *iniMover* y servirían para que las EFSM secundarias detectaran si la principal está en el estado NUEVA_R o en MOVIENDO, respectivamente. Las señales de entrada serían *finLeer* y *finMover* para indicar a la EFSM principal si las secundarias están en los estados LEÍDO y MOVIDO, respectivamente.

En general, pues, la implementación de PSM como conjunto de EFSM pasa por una etapa de construcción de las EFSM de las secuencias de acciones para cada estado-programa y la de la EFSM global, en la que hay que tener en cuenta que se debe añadir estados de espera y señales para la comunicación entre EFSM: una para poner en marcha el programa (*start*) y otra para indicar la finalización (*stop*).

Figura 24. Transformación de una PSM a una jerarquía de dos niveles de EFSM



En la figura 24 se ve este tipo de organización con un ejemplo sencillo que sigue al de la figura 16. El estado programa ESTADO_1 se desdobla en dos estados: ESTADO_1B ("B" de *begin*) y ESTADO_1E ("E" de *end*). En el primero, además de hacer la primera acción del programa, se activa la señal *start* para que la EFSM del programa correspondiente pase a ESTADO_11. Después de ponerlo en marcha, la máquina principal pasa al ESTADO_1E, en el que espera que la señal *stop* se ponga a 1. Las condiciones de salida del estado-programa ESTADO_1 son las mismas que las del ESTADO_1E, excepto que están multiplicadas lógicamente por *stop*, ya que no se hará la transición hasta que no se complete la ejecución del programa. Por su parte, la EFSM de control de programa, al llegar al último estado, activará la señal *stop*.

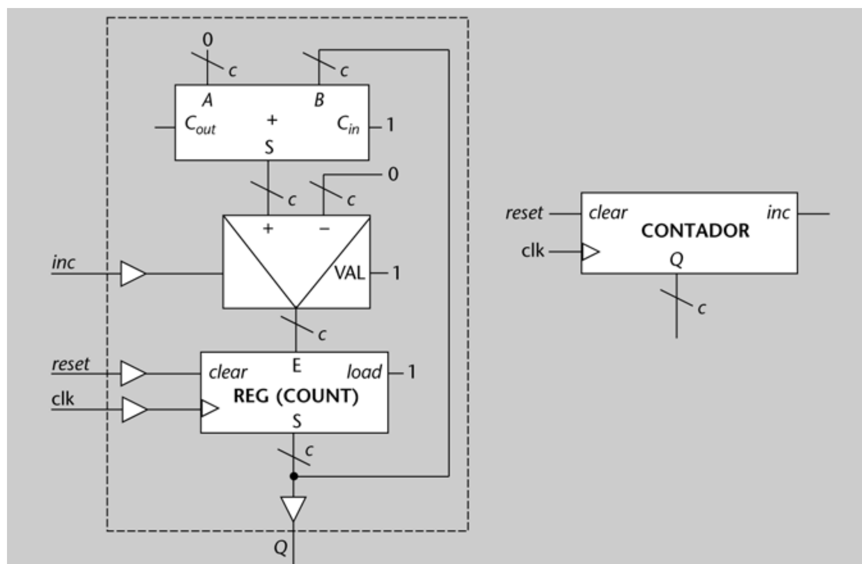
La materialización de estos circuitos es relativamente sencilla, ya que es suficiente con construir cada uno de las EFSM. Hay que recordar que la arquitectura de cada EFSM es de FSMD, con la unidad de control diferenciada de la unidad de proceso.

En los casos en los que la parte operacional sea compartida, como en el del ejemplo del controlador de velocidad, se puede optar por integrar las unidades de control en una sola. Esta parte de control unificada consiste en una FSM principal, que gobierna el conjunto y que activa un contador cada vez que se llega a un estado-programa (con más de una acción en secuencia). El cálculo del estado siguiente de esta FSM principal, con el contador activado, no tiene que variar hasta que se llegue al final de la cuenta para aquel estado. Es decir, mientras el contador no haya llegado al último valor que contar, el estado siguiente será el estado actual. El final de la cuenta coincide, evidentemente, con la compleción del programa asociado.

**Contador con señal de incremento**

Como ya se ha visto, los contadores son módulos que tienen muchos usos. Para la construcción de PSM con las EFSM principales y secundarias integradas, se puede utilizar cualquiera de los contadores que se han visto con anterioridad o uno más específico, uno que no necesite que se le dé el número hasta el cual tiene que contar (lo que se había denominado *M*). A cambio, deberá tener una señal de entrada específica que le indique si tiene que incrementar el valor de la cuenta o, por el contrario, ponerlo a cero.

Figura 25. Esquema del circuito contador con señal de incremento



Este contador es, de hecho, como un contador autónomo, en el que, en cada estado, puede pasar al estado inicial o al siguiente, según el valor de la señal de entrada de incremento, *inc*.

Hay que recordar que la distinción entre un tipo de módulo contador y otro se puede hacer viendo las entradas y salidas que tiene.

En la figura 26 hay un esquema de este modelo de construcción de PSM. En este caso, el circuito de cálculo de las salidas genera una interna, *inc*, que permite activar el contador. La cuenta se guarda en un registro específico (COUNT) del contador que se actualiza con un cero, si $inc = 0$, o con el valor anterior incrementado en una unidad, si $inc = 1$. Solo en los estados-programa que tengan más de una acción en secuencia se activará esta salida, que se retornará a cero cuando se acabe la ejecución.

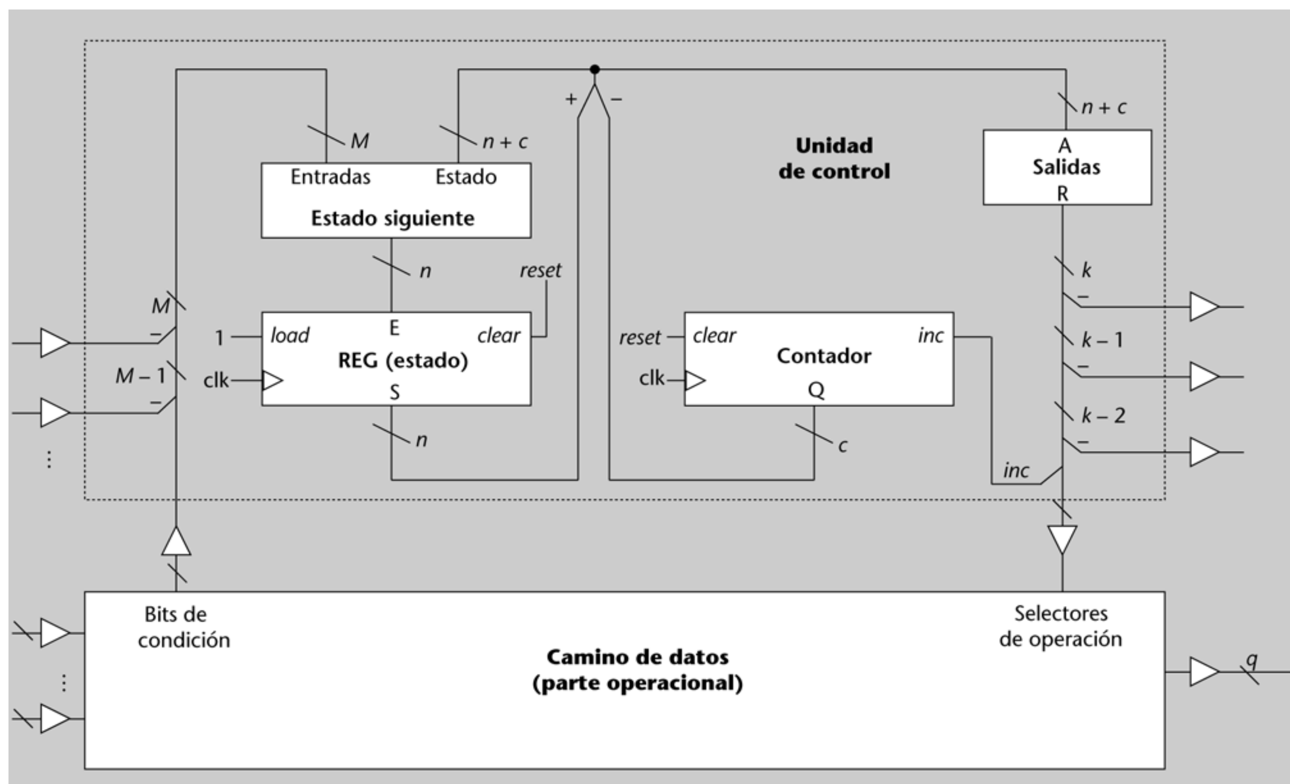
Así pues, la unidad de control está constituida por una máquina de estados compuesta de una FSM principal y distintas secundarias que comparten un único registro de estado, que es el registro interno del contador, COUNT. Eso es posible porque no están nunca activas al mismo tiempo. Consiguientemente, la máquina de estados que las engloba todas tiene un estado compuesto por el registro ESTADO y por el registro COUNT. Por este motivo, los buses hacia los bloques combinacionales de cálculo de estado siguiente y de cálculo de salidas son de $n + c$ bits.

**Contador de los estados-programa**

El contador de los estados-programa se ocupa de hacer la cuenta de la secuencia de operaciones que se realizan. Por eso se denomina ***contador de programa***. Como se verá más adelante, los procesadores también utilizan "contadores de programa".

Para la materialización de este tipo de PSM, como en el caso de cualquier máquina de estados, hay que hacer la tabla de transiciones y de salidas. Para el controlador de velocidad, la codificación de los estados se ha efectuado siguiendo la numeración binaria, dejando el código 0 para el estado inicial.

Figura 26. Arquitectura de una PSM con TOC y camino de datos común



Las entradas se corresponden con las señales de carga de nueva referencia (*ld_r*) y fin del periodo de un temporizador (*end*). El valor de *count* hace referencia al contenido del registro COUNT del contador de la unidad de control.

| **Estado actual** | | **Entradas** | | | **Estado⁺** | **Comentario** |
|---|---|---|---|---|---|---|
| **Símbolo** | $S_{1-0}$ | *ld_r* | *end* | *count* | $S_{1-0}$ | |
| INICIO | 00 | 0 | x | x | 00 | |
| INICIO | 00 | 1 | x | x | 10 | |
| MIDIENDO | 01 | 0 | 0 | x | 01 | |
| MIDIENDO | 01 | 0 | 1 | x | 11 | |
| MIDIENDO | 01 | 1 | x | x | 10 | |
| NUEVA_R | 10 | x | x | 00 | 10 | 1.er paso del programa |
| NUEVA_R | 10 | x | x | 01 | 10 | 2.o paso del programa |
| NUEVA_R | 10 | 0 | x | 10 | 01 | 3.er paso del programa y salida según entradas |
| NUEVA_R | 10 | 1 | x | 10 | 10 | |
| MOVIENDO | 11 | x | x | 00 | 11 | 1.er paso del programa |
| MOVIENDO | 11 | x | x | 01 | 11 | 2.o paso del programa |
| MOVIENDO | 11 | 0 | x | 10 | 01 | 3.er paso del programa y salida según entradas |
| MOVIENDO | 11 | 1 | x | 10 | 10 | |

La tabla de salidas asociadas a cada estado sencillo y a cada paso de un estado-programa es la que se muestra en la tabla siguiente.

| **Estado actual** | | **Paso de programa (*count*)** | **Salidas** | | | | |
|---|---|---|---|---|---|---|---|
| **Símbolo** | $S_{1-0}$ | | *inc* | *begin* | *SelOpU* | *SelOpT* | *SelOpW* |
| INICIO | 00 | x | 0 | 0 | 0 | 00 | 0 |
| MIDIENDO | 01 | x | 0 | 0 | 0 | 00 | 0 |
| NUEVA_R | 10 | 00 | 1 | 0 | 1 | 10 | 0 |
| NUEVA_R | 10 | 01 | 1 | 0 | 0 | 00 | 0 |
| NUEVA_R | 10 | 10 | 0 | 1 | 0 | 00 | 1 |
| MOVIENDO | 11 | 00 | 1 | 0 | 0 | 11 | 0 |
| MOVIENDO | 11 | 01 | 1 | 0 | 0 | 00 | 0 |
| MOVIENDO | 11 | 10 | 0 | 1 | 0 | 00 | 1 |

Cabe señalar que, para las secuencias de los estados-programa, hay que poner *inc* a 1 en todos los pasos excepto en el último. Las columnas encabezadas con “SelOp” son de selección de operación para las variables afectadas: $U$, $T$ y $W$, de izquierda a derecha en la tabla. La variable $A$ no aparece porque se deja que se actualice siempre con $M[T_{7-1} + T_0]$, que es el único valor que se le asigna en los dos programas. Una cosa similar pasa con $U$, que solo recibe el valor de la entrada $R$. En este caso, sin embargo, hay dos opciones: que $U$ tome un nuevo valor de $R$ o que mantenga el que tiene. De cara a la implementación, se ha

codificado *SelOpU* de manera que coincida con la señal de carga del registro correspondiente. La variable $T$ tiene tres opciones: mantener el contenido, cargar el de $R$ o el de $U + T - V$. Consiguientemente, *SelOpT* tiene tres valores posibles: 00, 10 y 11, respectivamente. Con esta codificación, el bit más significativo sirve como indicador de carga de nuevo valor para el registro correspondiente.

Finalmente, la variable $W$, que es la que proporciona la salida, puede tener tres valores siguientes posibles: 0 (INICIO), $W$ (MIDIENDO) y el cálculo del ancho (NUEVA_R y MOVIENDO). Para simplificar la implementación, dado que solo se pone a cero en el estado inicial y que se tiene que poner inmediatamente (la operación es $W = 0$, no $W^+ = 0$), se supone que el *reset* del registro asociado se hará cuando se ponga en el estado INICIO, junto con el de todos los otros registros de la máquina. Así, *SelOpW* solo debe discriminar entre si se mantiene el valor o si se actualiza y se puede hacer corresponder con la señal de carga del registro asociado.

Finalmente, hay que hacer una consideración en cuanto a los accesos a la memoria ROM: las direcciones son diferentes en el paso 1 y en el paso 2. La distinción se efectúa aprovechando la señal de *SelOpW*: cuando sea 1, la dirección será $T_{7-1}$ y cuando sea 0, $T_{7-1} + T_0$.

A partir de las tablas anteriores se puede generar el circuito. Hay que tener en cuenta que, con respecto a las funciones de las salidas, la tabla es incompleta, ya que hay casos que no se pueden dar nunca (el contador a 11) y que se pueden aprovechar para obtener expresiones mínimas para cada caso.

Siguiendo la arquitectura que se ha presentado, se puede construir el circuito secuencial correspondiente implementando las funciones de las tablas anteriores. El circuito está hecho a partir de las expresiones mínimas, en suma de productos, de las funciones.

El esquema del circuito resultante se presenta en la figura 27, organizado en dos bloques: el de control en la parte superior y el de procesamiento de datos en la inferior. Como se puede comprobar, es conveniente respetar la organización de la arquitectura de FSMD y separar las unidades de control y de proceso.

Antes de acabar, hay que señalar que se trata de un controlador de velocidad básico, que solo tiene en cuenta las situaciones más habituales.

**Puertas NOT**

Por simplicidad, los inversores o puertas NOT en las entradas de las puertas AND se sustituyen gráficamente por bolas. Es una opción frecuente en los esquemas de circuitos. Por ejemplo, las puertas AND de la figura siguiente implementan la misma función.

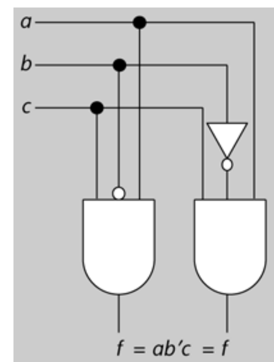
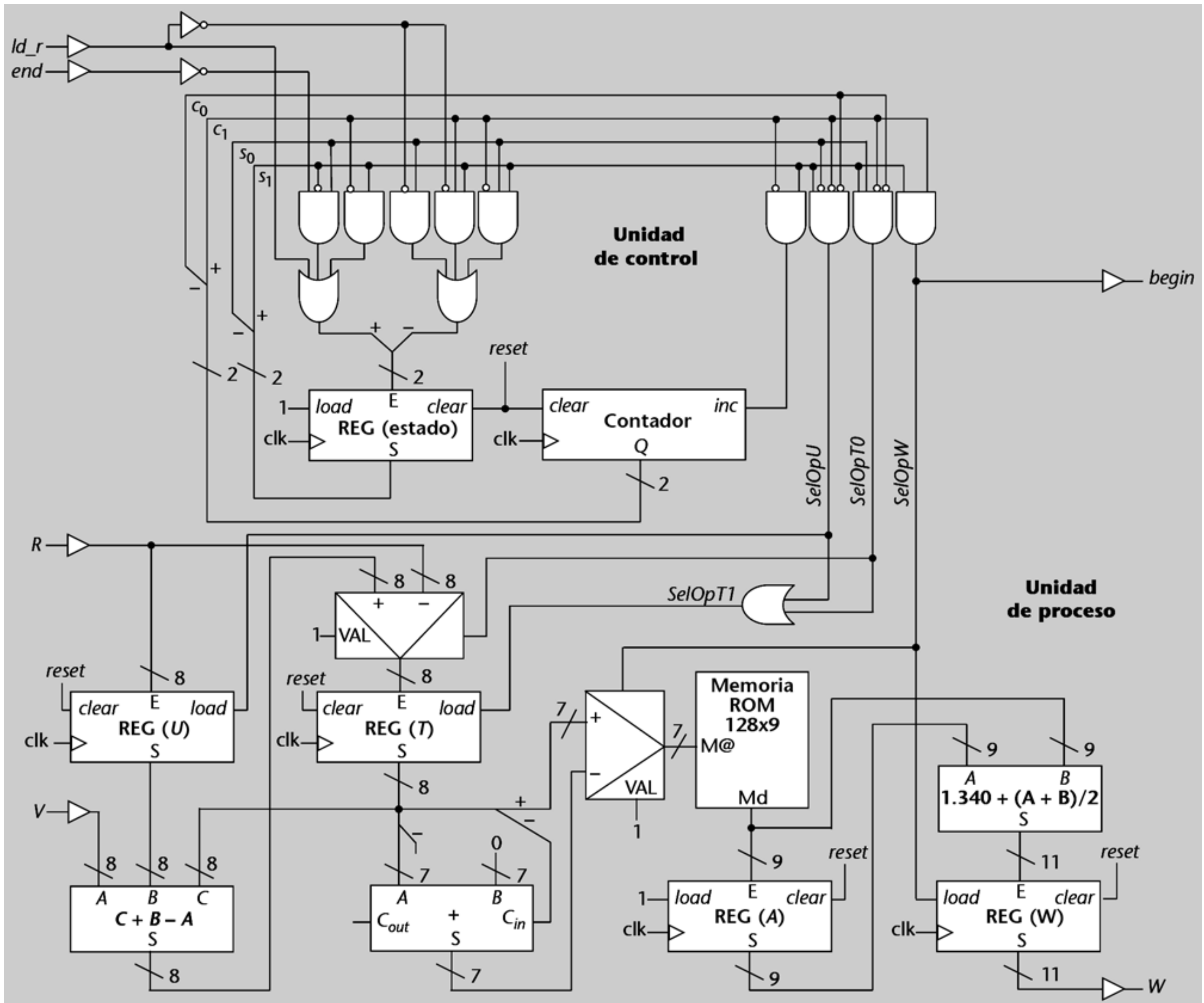


Figura 27. Circuito del control adaptativo de velocidad




## Actividades

**6.** Para completar el circuito del controlador de velocidad, diseñad los módulos combinacionales correspondientes a los cálculos $C + B - A$ y 1.340 + $(A + B)/2$. Prestad atención a los diferentes formatos de los números.

**7.** El modelo del controlador de velocidad que se ha visto recibe dos señales del controlador global: *ld_r* y *R*. La señal *ld_r* actúa exclusivamente como indicador de que el contenido de *R* ha cambiado y que es necesario que el controlador de velocidad cargue el nuevo valor de referencia. Por lo tanto, desde el punto de vista del controlador global, sería suficiente con efectuar un cambio en *R* para que el controlador de velocidad hiciera la carga de la nueva referencia y, de esta manera, no tendría que utilizar *ld_r*. Como el valor de la velocidad de referencia se guarda en la variable *U*, el controlador de velocidad puede determinar si hay cambios en el valor de la entrada *R* o no. Modificad el diagrama de la PSM para reflejar este cambio. ¿Cómo cambiaría el circuito?

### 1.4. Máquinas de estados algorítmicas

Las máquinas de estados que se han tratado hasta el momento todavía tienen el inconveniente de tener que representar completamente todas las posibles condiciones de transición. Ciertamente, se puede optar por suponer que aquellas que no se asocian a ningún arco de salida son de mantenimiento en el es-

tado actual. Con todo, todavía continúa siendo necesario poner de manera explícita los casos que implican transición hacia un estado diferente del actual.

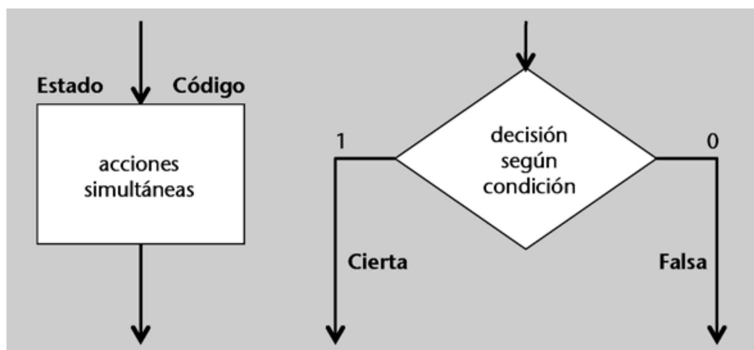
Visto desde otro punto de vista, los diagramas de estados tienen que ir acompañados, necesariamente, de las condiciones (entradas) y de las acciones (salidas) que se establecen para cada arco y nodo que tienen. Eso provoca que, cuando el número de entradas y salidas crece, resulten pesados de tratar. Las PSM resuelven el problema de las salidas cuando se trata de secuencias de acciones, pero siguen requiriendo que, para cada estado, se especifiquen todos los casos posibles de entradas y los estados siguientes para cada caso.

De hecho, en la mayoría de los estados, hay pocas salidas que cambian y las transiciones hacia los estados siguientes también dependen de unas pocas entradas. Los grafos de transiciones de estados se pueden representar de una manera más eficiente si se tiene en cuenta este hecho.

Las **máquinas de estados algorítmicas** o **ASM** (del inglés, *algorithmic state machines*) son máquinas de estados que relacionan las distintas acciones, que están ligadas a los estados, mediante condiciones determinadas en función de las entradas que son significativas para cada transición.

En los diagramas de representación de las ASM hay dos tipos de nodos básicos: las cajas de estado (rectangulares) y las de decisión (romboides). Las de estado se etiquetan con el nombre del estado y su código binario, habitualmente en la parte superior, tal como aparece en la figura 28. Dentro se ponen las acciones que se deben llevar a cabo en aquel estado. Son acciones que se efectúan simultáneamente, es decir, en paralelo. Las cajas de decisión tienen dos salidas, según si la condición que se expresa en su interior es cierta (arco etiquetado con el bit 1) o falsa (arco etiquetado con el bit 0).

Figura 28. Tipos de nodos en una máquina de estados algorítmica

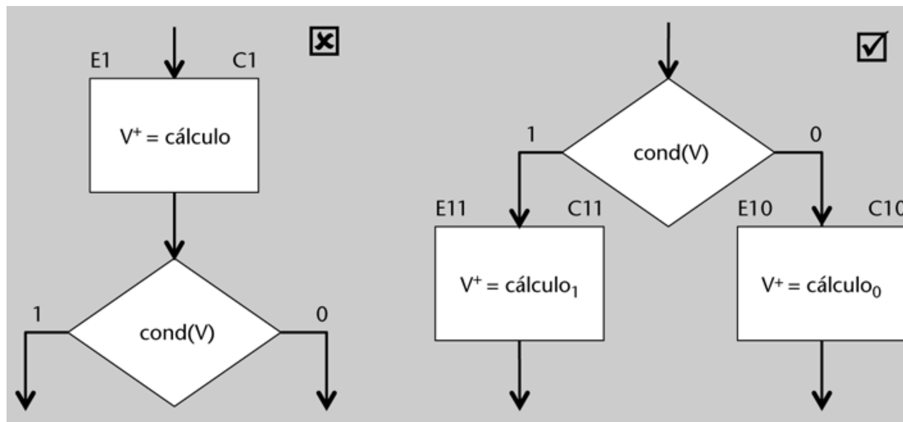



En comparación con los grafos que representan EFSM, las cajas de estados se corresponden a los nodos y las de decisión, a los arcos.

Hay que tener en cuenta, como con cualquier máquina de estados, que las condiciones que se calculan se hacen a partir del estado mismo y de los valores

almacenados en aquel momento en las variables de la misma máquina, pero no de los contenidos futuros. Para evitar confusiones, es recomendable que los diagramas de los ASM pongan primero las cajas de decisión y después las de estado y no al revés, tal como se ilustra en la figura 29.

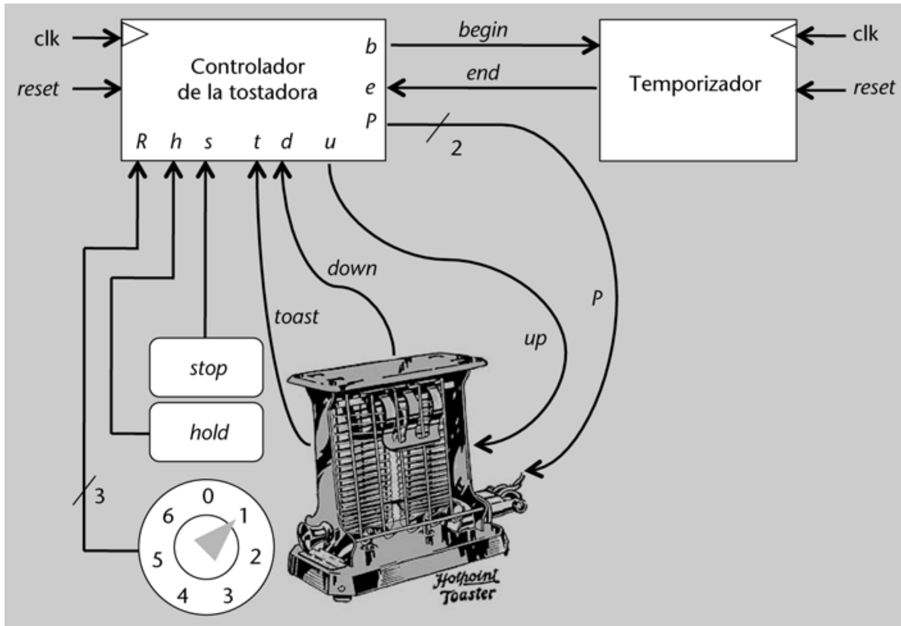
Figura 29. Formas desaconsejada y aconsejada de colocar estados con cálculos que afectan a bits de condición




Para ver cómo representar el funcionamiento de un controlador con una ASM, se estudiará el caso para una tostadora de pan. Se trata de un "dispositivo" relativamente sencillo que tiene dos botones: uno (*stop*) para interrumpir el funcionamiento y hacer saltar la tostada y otro (*hold*) para mantenerla caliente dentro hasta que se apriete el botón anterior. Además, tiene una rueda que permite seleccionar 6 grados diferentes de tostado y que también puede estar apagada (posición 0). Para ponerla en marcha, solo se requiere poner el selector en una posición diferente de 0 y hacer bajar la rebanada de pan. Hay un sensor para indicar cuándo se baja el soporte del pan (*down*) y otro para detectar la presencia de la rebanada (*toast*). En cuanto a las salidas del controlador, hay una para hacer subir la tostada (*up*) y otra para indicar la potencia de los elementos calefactores (*P*). Para tener en cuenta el tiempo de cocción, hay un temporizador que sirve para hacer una espera de un periodo prefijado. Así, el proceso de tostar durará tantos periodos de tiempo como número de posición tenga el selector.

El esquema de la tostadora se muestra en la figura 30 (la tostadora representada es un modelo de principios del siglo XX que, evidentemente, tenía un funcionamiento muy diferente del presentado). En la figura también aparece el nombre de las señales para el controlador. Hay que tener presente que la posición de la rueda se lee a través de la señal *R* de tres bits (los valores están en el rango [0, 6]) y que la potencia *P* de los elementos calefactores se codifica de manera que 00 los apaga totalmente, 10 los mantiene para dar un calor mínimo de mantenimiento de la tostada caliente y 11 los enciende para irradiar calor suficiente para tostar. Con *P* = 11, la rebanada de pan se tostará más o menos según *R*. De hecho, la posición de la rueda indica la cantidad de periodos de tiempo en los que se hará la tostada. Cuanto más valor, más lapsos de tiempo y más tostada quedará la rebanada de pan. Este periodo de tiempo se controlará con un temporizador, mientras que el número de periodos se contará internamente con el ASM correspondiente, mediante una variable auxiliar de cuenta, *C*.

Figura 30. Esquema de bloques de una tostadora




En este caso, el elevado número de entradas del controlador ayuda a mostrar las ventajas de la representación de su comportamiento con una ASM. En la figura 31 se puede observar el diagrama completo.

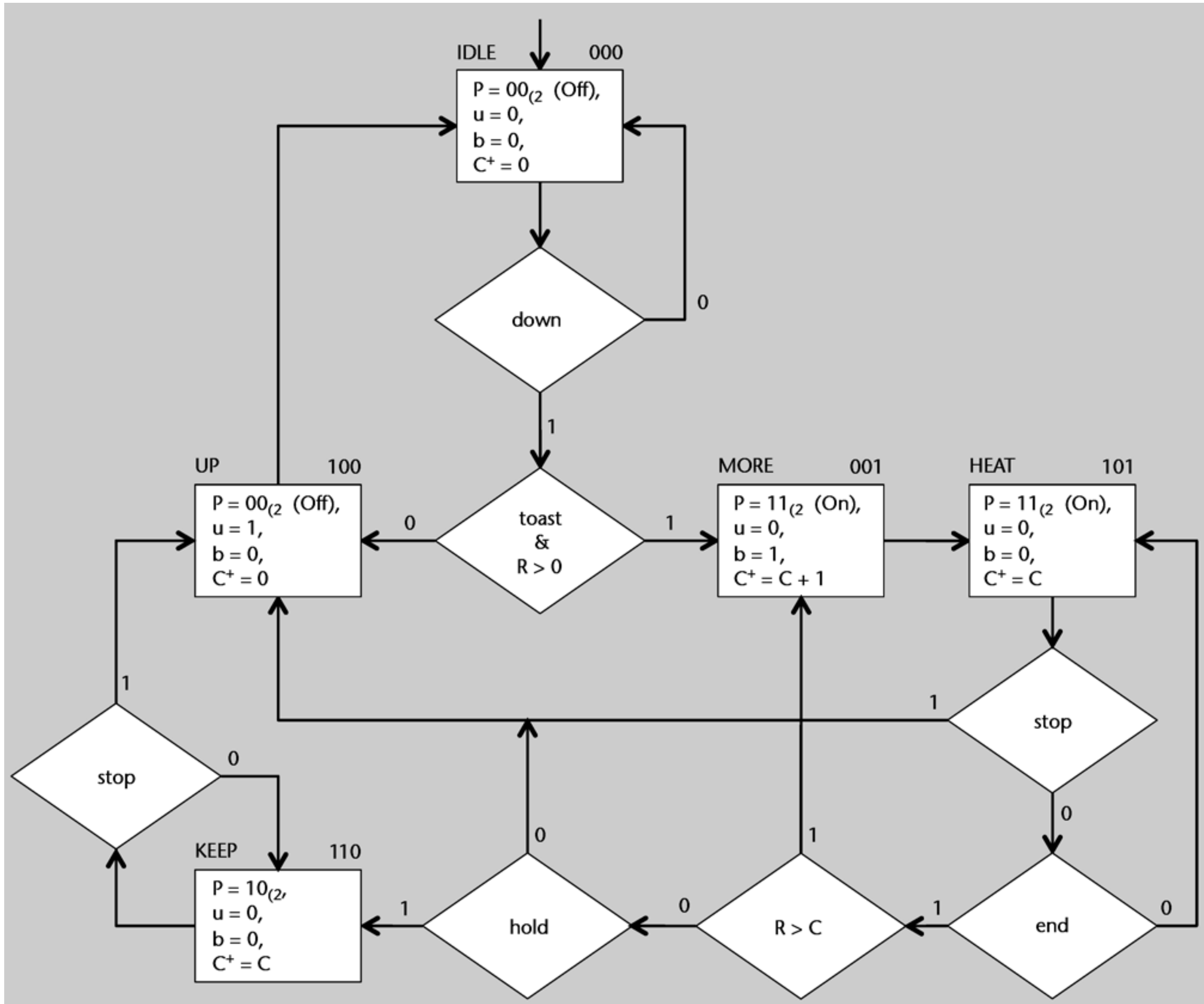
Es recomendable que las acciones y las condiciones que se describen en las cajas del diagrama de estados sean lo más generales posible. De esta manera son más inteligibles y no quedan vinculadas a una implementación concreta.

A modo de ejemplo, en las cajas de decisión aparece solo la indicación textual de las condiciones que se han de cumplir. Por comparación, en las cajas de estado, las acciones se describen directamente con los cambios en las señales de control y variables, tal como aparecen simbolizadas en la parte interior del módulo del controlador de la tostadora en la figura 30. Se puede comprobar que, en un caso un poco más complejo, las acciones serían más difíciles de interpretar. En cambio, las condiciones resultan más inteligibles y, además, se pueden trasladar fácilmente hacia expresiones que, a su vez, se pueden materializar de manera bastante directa.

Hay que tener presente la distinción entre variables y señales de salida: los valores de las señales de salida son los que se calculan en cada caja de estado, mientras que los valores de las variables se mantienen durante todo el estado y solo se actualizan con el resultado de las expresiones que hay en la caja de estado cuando se pasa al estado siguiente. !

El estado inicial de la ASM es IDLE, en el que los radiadores de la tostadora están apagados ($P = 00$), no se hace ninguna acción (*up* o *begin*), ni hacer saltar la rebanada de pan ($u = 0$), ni poner en marcha el temporizador ($b = 0$) y, además, se pone a cero la variable $C$, lo que tendrá efecto en el estado siguiente, tal como se indica con el símbolo *más* en posición de superíndice.

Figura 31. Diagrama de una ASM para el controlador de una tostadora




En el estado IDLE se efectúa la espera de que se ponga una rebanada de pan en la tostadora. De aquí que en el arco de salida la condición que se tiene que satisfacer sea *down*. Si no se ha puesto nada, la máquina se queda en el mismo estado. Si no, pasa a comprobar que, efectivamente, se haya insertado una rebanada de pan (*toast*) y que se haya puesto la rueda en una posición que no sea del apagado ($R > 0$). En función del resultado de esta segunda condición, la máquina pasará al estado UP o al estado MORE. En el primer caso, se hará subir el soporte del pan (o bien no se había metido nada o bien había una rebanada pero la rueda estaba en la posición 0) y se retornará, finalmente, al estado inicial de espera. En el segundo caso, se pondrán en marcha los calefactores para iniciar la torrefacción. También se pondrá el contador interno de intervalos de tiempo a 1 y se activará un temporizador para que avise del momento en el que se acabe el periodo de tiempo del intervalo actual.

Es interesante observar cómo, en las ASM, las cajas de estado solo tienen un arco de salida que se va dividiendo a medida que atraviesa cajas de condición. Por ejemplo, el arco de salida de IDLE tiene tres destinos posibles: IDLE, MORE y UP, según las condiciones que se satisfacen. De manera diferente a como se debería hacer en los modelos de PSM y ESFM, en los que un estado similar debería tener tres arcos de salida con expresiones de condición completas para cada arco.

Tal como se ha comentado, el número de intervalos de tiempo que tienen que transcurrir para hacer la tostada viene determinado por la intensidad de la torrefacción que se regula mediante la posición de la rueda de control, que puede variar de 1 a 6. En el estado MORE se incrementa el contador interno, $C$, que tiene la función de contar cuántos intervalos de tiempo lleva en funcionamiento.

De este estado se pasa al de tostar (HEAT), en el que se queda hasta que se apriete el botón de parada (*stop*) o se acabe el periodo de tiempo en curso (*end*). En el primer caso, pasará a UP para hacer subir la tostada tal como esté en aquel momento y retornará, finalmente, a IDLE. En el segundo caso, comprobará cuántos periodos han de pasar todavía hasta conseguir la intensidad deseada de torrefacción. Si $R > C$, vuelve a MORE para incrementar el contador interno e iniciar un nuevo periodo de espera.

El arco de salida de HEAT todavía tiene una derivación más. Si ya se ha completado la torrefacción ($R > C$), hay que hacer subir la tostada (ir a UP) a menos que el botón de mantenimiento (*hold*) esté pulsado. En este caso, se pasa a un estado de espera (KEEP) que deja los elementos calefactores a una potencia intermedia que mantenga la tostada caliente. Al pulsar el botón de paro, se pasa a UP para hacer saltar la tostada y apagar la tostadora.

Es importante tener en cuenta que se ha seguido la recomendación ejemplificada en la figura 29: la caja de condición en función de una variable ($R > C$) precede a una de estado que la modifica (MORE).

Para la materialización de la ASM hay que representar las expresiones de todas las cajas con fórmulas lógicas. Por este motivo conviene hacer lo mismo que se ha hecho con las cajas de estado: sustituir los nombres de las condiciones por las expresiones lógicas correspondientes. En la tabla siguiente, se muestra esta relación.

| Condición | Expresión lógica |
|---|---|
| *down* | $d$ |
| *toast* & $R > 0$ | $t \cdot (R > 0)$ |
| *stop* | $s$ |
| *end* | $e$ |
| $R > C$ | $(R > C)$ |
| *hold* | $h$ |

Para obtener las funciones que calculan el estado siguiente, se puede optar por hacerlo a partir de la tabla de verdad de las funciones de transición. En el caso de las ASM, hay otra opción en la que el estado siguiente se calcula a partir de los resultados de la evaluación de las condiciones que hay en las cajas de decisión. La idea del método es que la transición de un estado a otro se lleva a

cabo cambiando solo aquellos bits que difieren de un código de estado al otro, según las evaluaciones de las condiciones de las cajas que los relacionan.

Con este segundo método, la construcción de las expresiones lógicas para el cálculo del estado siguiente es directa desde las ASM. Para que sea eficiente, es conveniente que la codificación de los estados haga que dos cajas de estado vecinas tengan códigos en los que cambian el mínimo número posible de bits.

**Cálculo del estado siguiente a partir de las diferencias entre códigos de estados**

A modo ilustrativo, se supone el ejemplo siguiente: una ASM está en el estado ($s_3$, $s_2$, $s_1$, $s_0$) = 0101 y, con las entradas $a = 1$ y $b = 1$, debe pasar al estado 0111. En la manera convencional, las funciones del estado siguiente, representadas en suma de productos, tendrían que incluir el término $s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b$, es decir:

$$
\begin{aligned}
s^+_3 &= s^*_3 \\
s^+_2 &= s^*_2 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b \\
s^+_1 &= s^*_1 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b \\
s^+_0 &= s^*_0 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b
\end{aligned}
$$

donde $s^*_i$ hace referencia a los otros términos producto de la función correspondiente, que son los otros 1 que tiene en la tabla de verdad asociada. A la expresión de $s^+_3$ no se le añade el término anterior porque la función no es 1 en este caso de estado actual y condiciones.

Si solo se tuvieran en cuenta los bits que cambian, las funciones que calculan el estado siguiente serían del tipo:

$$
\begin{aligned}
s^+_3 &= s_3 \oplus c_3 \\
s^+_2 &= s_2 \oplus c_2 \\
s^+_1 &= s_1 \oplus c_1 \\
s^+_0 &= s_0 \oplus c_0
\end{aligned}
$$

donde $c_i$ hace referencia a la función de cambio de bit. Hay que tener en cuenta que:

$$
\begin{aligned}
s_i \oplus 0 &= s_i \\
s_i \oplus 1 &= s'_i
\end{aligned}
$$

Siguiendo con el ejemplo, para construir las expresiones de las funciones de cambio $c_i$, se tiene que ver qué bits entre el código de estado origen y destino cambian y añadir el término correspondiente a la suma de productos que toque. En este caso, solo cambia el bit que hay en la posición 1:

$$
\begin{aligned}
c_3 &= c^*_3 \\
c_2 &= c^*_2 \\
c_1 &= c^*_1 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b \\
c_0 &= c^*_0
\end{aligned}
$$

donde $c^*_i$ representa la expresión en sumas de productos para los otros cambios en el bit iésimo del código de estado. De esta manera, siempre que los códigos de estados vecinos sean próximos (es decir, que no difieran en demasiados bits), las funciones de cálculo de estado siguiente tienen una representación muy directa y sencilla, con pocos términos producto.

Para verlo más claro, en la figura 32 hay una ASM y el circuito que realiza el cálculo del estado siguiente. Como todos los términos que se suman a las funciones de cambio incluyen el producto con el estado actual, se opta por utilizar multiplexores que, además, ayudan a hacer la suma final como todos los términos. (En este caso, como solo hay un arco de salida para el estado *S*0, los multiplexores solo sirven para hacer el producto para $s'_1 \cdot s'_0$). Para cada estado, que el bit cambie o no, depende del código del estado de destino. En el ejemplo, solo se puede pasar del estado *S*0 (00) a los *S*1 (01), *S*2 (10) y *S*3 (11). Y los cambios consisten en poner a 1 el bit que toque. Para que cambie el bit menos significativo (LSB) del estado, *C*0 debe ser 0 (rama derecha de la primera caja de decisión) o *C*1 y *C*0 deben ser 1 al mismo tiempo (ramas izquierdas del esquema). Para que cambie el MSB del estado, se ha de cumplir que $C1' \cdot C0$ o que $C1 \cdot C0$. Es decir, es suficiente con que *C*0 sea 1. En la figura el circuito no está minimizado para ilustrar bien el modelo de construcción del cálculo del estado siguiente.

Figura 32. Modelo constructivo de la unidad de control de una ASM

En el circuito de la figura 32 se han dejado los dos multiplexores que sirven para hacer el producto lógico entre los estados origen y las condiciones ligadas a los arcos de salida que llevan a estados destino que tienen un código en el que el bit correspondiente es diferente del de los estados origen. Como en el ejemplo solo se han tratado los arcos de salida del estado *S*0, solo se aprovecha la entrada 0 de los multiplexores. Si las otras cajas de estado tuvieran arcos de salida, entonces las condiciones correspondientes serían las entradas de los multiplexores en las posiciones 1, 2 o 3.

Las unidades de control que siguen esta arquitectura se pueden construir sin necesidad de calcular explícitamente las funciones de cálculo del estado siguiente, algo muy ventajoso para los casos en los que el número de entradas es relativamente grande.

Para este tipo de cálculos, la codificación de los estados debe realizarse de manera que entre dos estados vecinos (es decir, que se pueda ir de uno al otro sin que haya ningún otro entre medio) cambie el mínimo número posible de bits.

Para construir el circuito secuencial de la tostadora, que se corresponde con la ASM de la figura 31, primero hay que obtener las ecuaciones que calculan el estado siguiente. Para ello, se sigue la arquitectura que se acaba de explicar.

Para hacerlo, es suficiente con determinar las expresiones para las funciones de cambio, ($c_2$, $c_1$, $c_0$). Una manera de proceder es, estado por estado, mirar qué términos se deben añadir, según el estado destino y los bits que cambian. Por ejemplo, desde IDLE (000) se puede ir a IDLE (000), MORE (001) o UP (100). De un estado a él mismo no se añade ningún término a la función de cambio porque, obviamente, no cambia ningún bit del código de estado. Del estado IDLE a MORE cambia el bit en posición 0, por lo tanto, hay que añadir el término correspondiente ($[s'_2s'_1s'_0] \cdot dtr$) a la función de cambio $c_0$. A las otras funciones no hay que añadirles nada. Finalmente, de IDLE a UP solo cambia el bit más significativo y hay que añadir el término $[s'_2s'_1s'_0] \cdot d(t' + r')$ a la función de cambio $c_2$. Como todos los términos, una parte la constituye la que es 1 para el estado origen y la otra, el producto de las expresiones lógicas de las condiciones que llevan al de destino. En este caso, que $d = 1$ y que $(t \cdot r = 0)$, es decir, que se cumpla que $d \cdot (t \cdot r)' = 1$ o, lo que es lo mismo, que $d(t' + r') = 1$.

Al final del proceso, las expresiones resultantes son las siguientes.

$$s^+_2 = s_2 \oplus ([s'_2s'_1s'_0] \cdot d(t' + r') + [s'_2s'_1s_0] \cdot 1 + [s_2s'_1s'_0] \cdot 1 + [s_2s'_1s_0] \cdot s'er)$$
$$s^+_1 = s_1 \oplus ([s_2s'_1s_0] \cdot s'ehr' + [s_2s_1s'_0] \cdot s)$$
$$s^+_0 = s_0 \oplus ([s'_2s'_1s'_0] \cdot dtr + [s_2s'_1s_0] \cdot (s + er'))$$

donde la señal $r$ representa el resultado de comprobar si $R > C$, que es equivalente a comprobar $R > 0$ en la caja de decisión correspondiente porque solo se puede llegar del estado IDLE, donde $C$ se pone en 0. Hay que tener en cuenta que, de hecho, $C$ se pone a 0 al principio del estado siguiente y que el valor actual en IDLE podría no ser 0 si el estado inmediatamente anterior hubiera sido UP. Por ello, es necesario que se haga $C^+ = 0$ también en UP.

La parte operacional solo debe materializar los pocos cálculos que hay, que son:

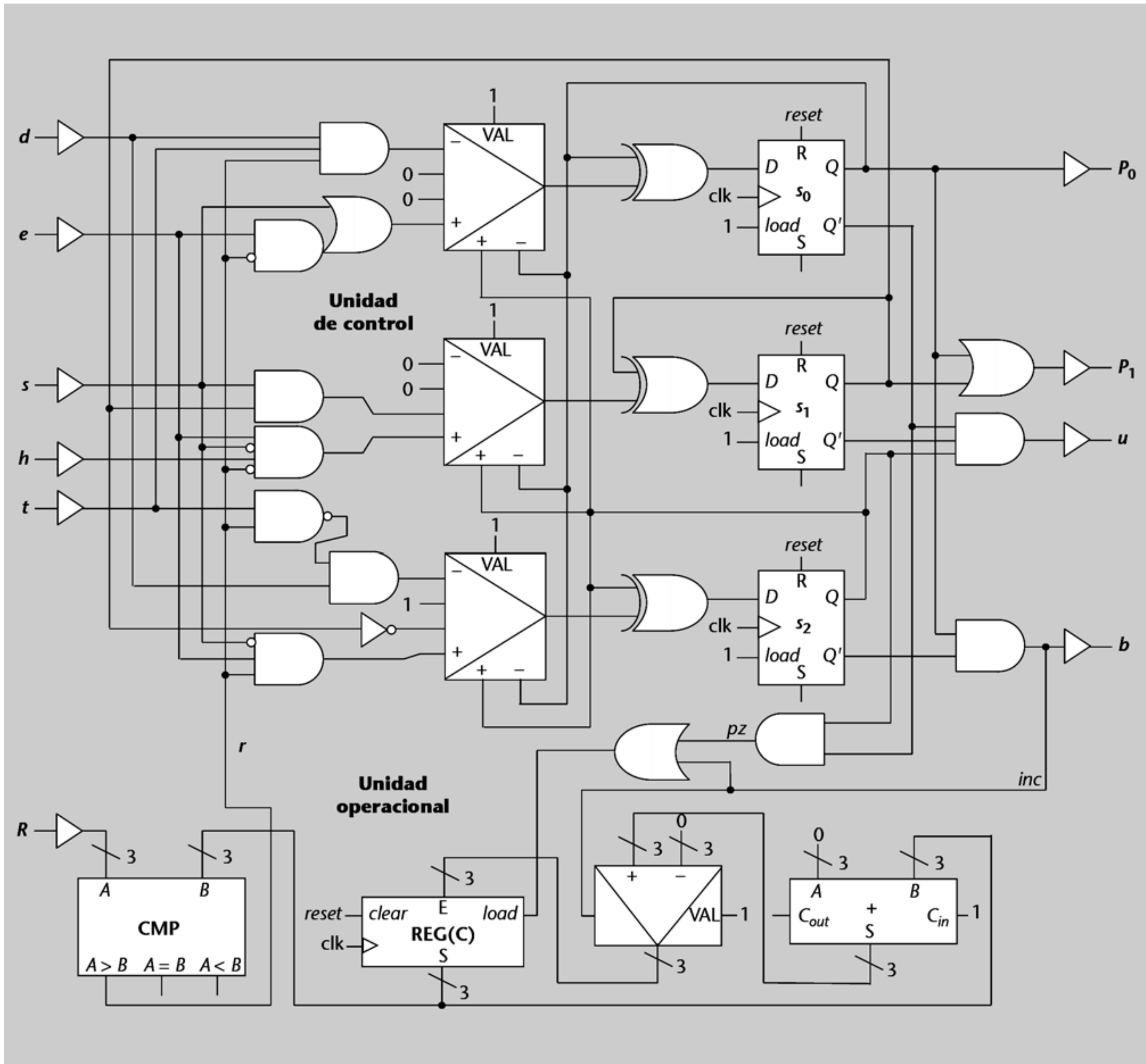
- Para un registro de cuenta $C$, $C^+ = 0$ y $C^+ = C + 1$, que se seleccionarán con las señales *inc*, de incremento, y *pz*, de poner a cero.
- La comparación con la referencia $R$, $R > C$, que da el bit de condición $r$.

Las funciones de salida (*inc*, *pz*, *u*, *b*, *P*) se calculan a partir del estado y son las siguientes:

$$inc = s'_2\, s'_1s_0 \text{ \{= MORE\}}$$
$$pz = s'_2\, s'_1s'_0 + s_2s'_1s'_0 \text{ \{= IDLE + UP\}}$$
$$u = s_2s'_1s'_0 \text{ \{= UP\}}$$
$$b = s'_2\, s'_1s_0 \text{ \{= MORE\}}$$
$$p_1 = s'_2\, s'_1s_0 + s_2s'_1s_0 + s_2s_1s'_0 \text{ \{= MORE + HEAT + KEEP\}}$$
$$p_0 = s'_2\, s'_1s_0 + s_2s'_1s_0 \text{ \{= MORE + HEAT\}}$$

Con todo, el esquema del controlador de la tostadora es el que se muestra en la figura 33.

Figura 33. Esquema del circuito del controlador de una tostadora




El circuito incluye una optimización basada en aprovechar como *don't-cares* los códigos de estado que no se utilizan, que son: 010, 011 y 111. Así pues, para saber si la máquina está en el estado 000, es suficiente con mirar los dos bits de los extremos, ya que 010 no se puede dar nunca. De manera similar, eso sucede con 001 y 101 respecto a los códigos irrelevantes 011 y 111, respectivamente. En el caso de los estados 100 y 110, hay que introducir una mínima lógica de diferenciación. Eso vale la pena a cambio de utilizar multiplexores de selección de estado de orden 2 en lugar de los de orden 3.

En la parte inferior de la figura se observa la unidad de procesamiento, con el comparador para calcular *r* y también cómo se han implementado los cálculos para la variable *C*. En este caso, el multiplexor selecciona si el valor en el estado siguiente de la variable, $C^+$, debe ser 0 o $C + 1$ y la señal de carga del registro correspondiente se utiliza para determinar si tiene que cambiar, caso que se da cuando *inc* + *pz*, o si ha de mantener el contenido que ya tenía.

## Actividades

**8.** Construid la tabla de verdad de las funciones de transición del controlador de la tostadora.

**9.** Modificad la ASM de manera que, cuando se introduzca una rebanada de pan, no se expulse automáticamente si $R = 0$. En otras palabras, que pase a un estado de espera (WAIT) hasta que $R > 0$ o hasta que se apriete *stop*.

En general, la materialización de una máquina de estados se lleva a cabo partiendo de aquella representación que sea más adecuada. Las FSM que trabajan con señales binarias son más sencillas de implementar, pero es más dificultoso ver la relación con la parte operacional. Las EFSM solucionan este problema integrando cálculos en el modelo de sistema. Según cómo, sin embargo, es conveniente empaquetar estas secuencias de cálculo (PSM) y simplificar las expresiones de las transiciones (ASM).

# 2. Máquinas algorítmicas

Las máquinas de estados sirven para representar el comportamiento de los circuitos de control de una multitud de sistemas bien diferentes. Pero no son tan útiles a la hora de representar los cálculos que se han de llevar a cabo en ellas, sobre todo, si gran parte de la funcionalidad del sistema consiste en eso mismo. De hecho, el comportamiento de los sistemas que hacen muchos cálculos o, en otras palabras, que hacen mucho procesamiento de la información que reciben se puede representar mejor con algoritmos que con máquinas de estados.

En esta parte se tratará de la materialización de sistemas en los que el procesamiento de los datos tiene más relevancia que la gestión de las entradas y salidas de estos. Dicho de otra manera, son sistemas que implementan algoritmos o, si se quiere, máquinas algorítmicas.

## 2.1. Esquemas de cálculo

Al diseñar máquinas de estados con cálculos asociados no se ha tratado la problemática ligada a la implementación de aquellos que son complejos.

En general, en un sistema orientado al procesamiento de la información, las expresiones de los cálculos pueden estar formadas por una gran cantidad de operandos y de operadores. Además, entre los operadores se pueden encontrar los más comunes y simples de materializar, como los lógicos, la suma y la resta, pero también productos, divisiones, potenciaciones, raíces cuadradas, funciones trigonométricas, etcétera.

Para materializar estos cálculos complejos, es necesario descomponerlos en otros más sencillos. Una manera de hacerlo es transformarlos en un programa, igual a como se ha visto para las máquinas de estados-programa.

Según cómo, sin embargo, es conveniente representar el cálculo de manera que sea fácil analizarlo de cara a obtener una implementación eficiente del camino de datos correspondiente, lo que no se ha tratado en el tema de las PSM.

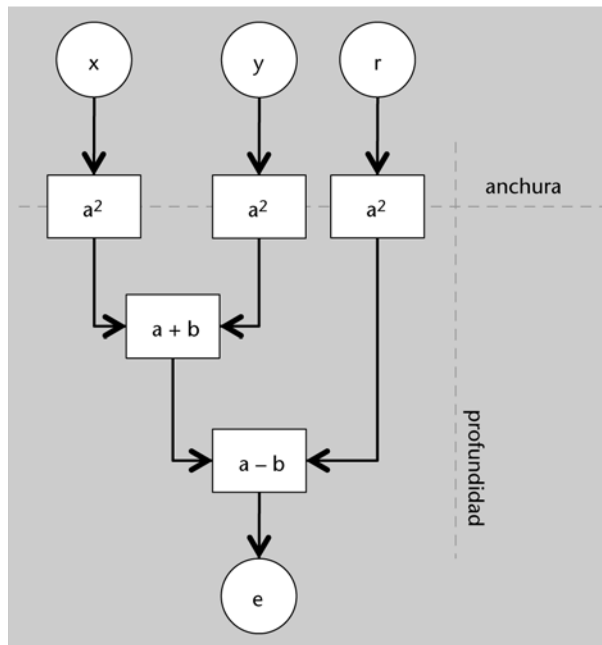
Los esquemas de cálculo son, de hecho, representaciones gráficas de cálculos más o menos complejos que se manipulan de cara a obtener una que sea adecuada para las condiciones del problema en el que se tengan que hacer. Unas veces interesará que el cálculo se haga muy rápido y otras, que se realice con el mínimo número de recursos de cálculo posible.

En un esquema de cálculo hay dos tipos de nodos: los que hacen referencia a los operandos (círculos) y los que hacen referencia a los operadores (rectángu-

los). La relación entre unos y otros se señala mediante arcos (flechas) que indican el flujo de los datos. Los operadores se suelen describir en términos de operandos genéricos, no vinculados a un cálculo específico. (En los ejemplos que aparecen a continuación se especifican con operandos identificados con las letras *a*, *b*, *c*...).

Por ejemplo, una expresión como $e = x^2 + y^2 - r^2$ se puede representar mediante el esquema de cálculo de la figura 34.

Figura 34. Grafo de un esquema de cálculo




En el esquema de cálculo anterior se puede observar la precedencia entre las operaciones, pero también otras cosas: la **anchura del cálculo**, que hace referencia al número de recursos de cálculo que se utilizan en una etapa determinada, y la **profundidad del cálculo**, que hace referencia al número de etapas que hay hasta obtener el resultado final.

Los recursos de cálculo son los elementos (habitualmente, bloques lógicos o módulos combinacionales) que llevan a cabo las operaciones. Las etapas en un cálculo quedan constituidas por todas aquellas operaciones que se pueden hacer de manera concurrente (en paralelo) porque los resultados son independientes. En la figura 34, hay tres: una para el cálculo de los cuadrados, otra para la suma, y la final, para la resta.

Puede suceder que haya operaciones que se puedan resolver en distintas etapas: en el caso de la figura 34, el cálculo de $r^2$ se puede hacer en la primera o en la segunda etapa, por ejemplo.

La profundidad mínima del cálculo es el número mínimo de etapas que puede tener. Cuanto mayor es, más complejo es el cálculo. En particular, si se supone

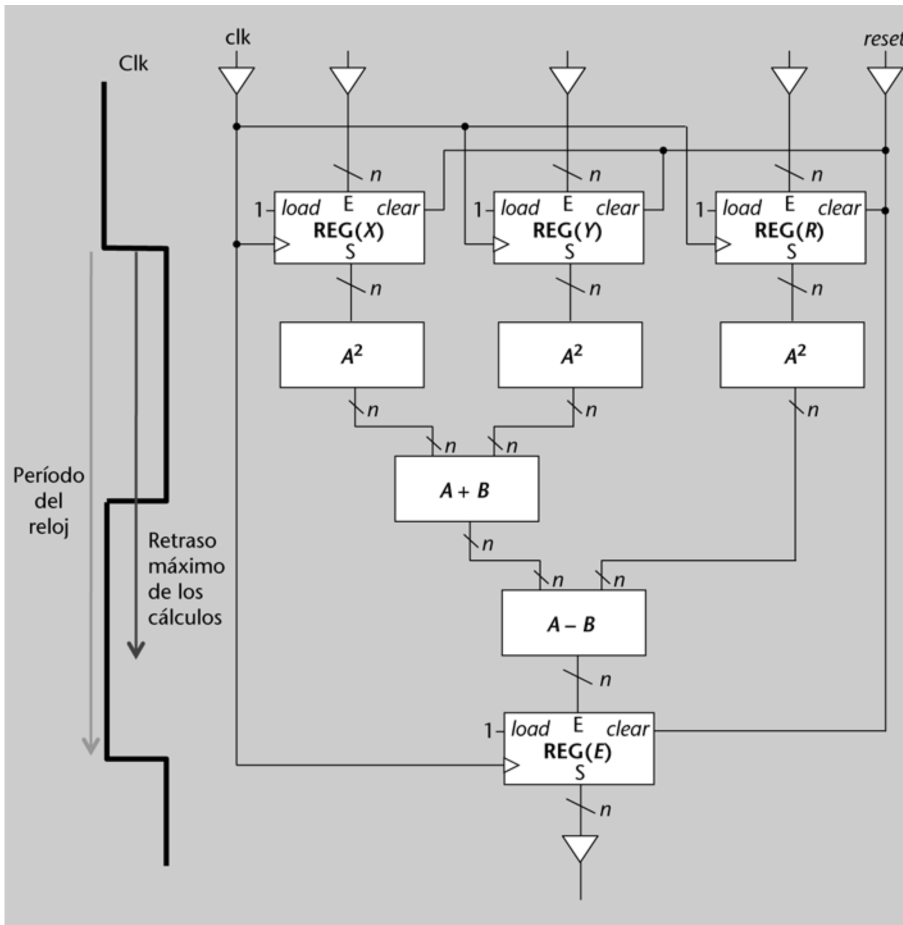
que los operandos se almacenan en registros, igual que el resultado, el cálculo se ve como una transferencia entre registros. Para el ejemplo anterior, sería:

$$E^{+} = X^2 + Y^2 - R^2$$

Siguiendo con esta suposición, los cálculos muy profundos necesitarán un tiempo de ciclo del reloj muy grande (lo podéis ver en la figura 35). Como se ha dicho, el tiempo de procesamiento depende de la profundidad del cálculo, ya que, cuantos más niveles de operadores, más etapas de bloques lógicos combinacionales deberán “atravesar” los datos para llegar a producir el resultado final. De hecho, se puede atribuir un tiempo de retraso a cada operador, de modo similar a como se hace con las puertas lógicas, y determinar el retraso que introduce un determinado cálculo. Si, como se ha supuesto, los datos de entrada provienen de registros y el resultado se almacena en otro registro, el periodo de reloj que los controla debe ser mayor que el retraso de los cálculos para obtener el resultado que se ha de cargar.

Eso puede ir en detrimento del rendimiento del circuito en caso de que haya cálculos muy complejos y profundos, y otros más sencillos, ya que el periodo del reloj se debe ajustar al peor caso.

Figura 35. Circuito de un esquema de cálculo con cronograma lateral




Por otra parte, los cálculos que tengan una gran amplitud de cálculo se deben llevar a cabo con muchos recursos que pueden no aprovecharse plenamente

cuando se evalúen otras expresiones, lo que también tiene una influencia negativa en la materialización de los circuitos correspondientes: son tan grandes como para cubrir las necesidades de la máxima amplitud de cálculo, aunque no la necesiten la mayor parte del tiempo.

En los próximos apartados se verá cómo resolver estos problemas y cómo implementar los circuitos correspondientes.

### 2.2. Esquemas de cálculo segmentados

Los esquemas de cálculo permiten manipular, de manera gráfica, las operaciones y los operandos que intervienen en un cálculo de cara a optimizar algún aspecto de interés. Habitualmente, se trata de minimizar el uso de recursos, el tiempo de procesamiento, o de llegar a un compromiso entre los dos factores.

Para reducir el tiempo de procesamiento de un cálculo concreto, hay que ajustarse a su profundidad mínima. Con todo, este tiempo debe ser inferior al periodo del reloj que marca las cargas de valores en los biestables del circuito. Eso hace que se tengan que tener en cuenta todas las operaciones que se pueden hacer y que aquellas que sean demasiado complejas y necesiten un tiempo relativamente mayor que las otras, se tengan que separar en varias partes o segmentos. Es decir, **segmentar** el esquema de cálculo correspondiente.

Entre segmento y segmento se tienen que almacenar los datos intermedios. Esto genera que las implementaciones de los esquemas segmentados necesiten registros auxiliares y, por lo tanto, sean más grandes. A cambio, el circuito global puede trabajar con un reloj más rápido.

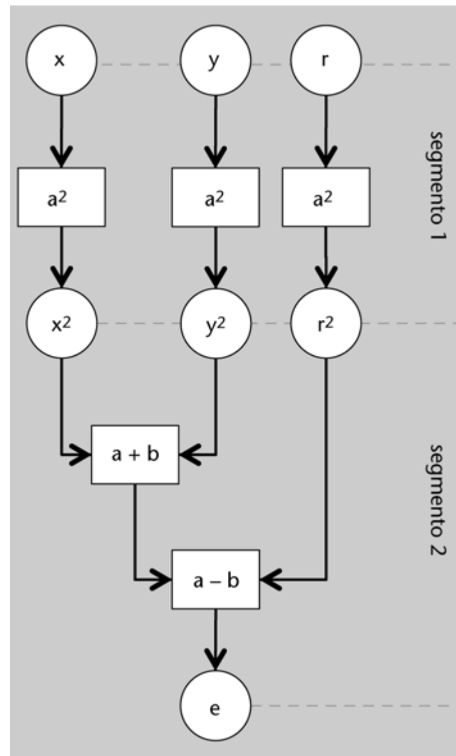
En la figura 36 aparece una segmentación del esquema de cálculo que se ha tomado como ejemplo en el apartado anterior. En el primer segmento se calculan los cuadrados de todos los datos y en el segundo se hacen las operaciones de suma y de resta. Para efectuar este cálculo se necesitan dos ciclos de reloj y, de manera directa, tres registros adicionales extras: uno para cada cuadrado.

Si se tienen que encadenar dos o más cálculos parciales seguidos, esta estructura tiene la ventaja de poderlos empezar en periodos de reloj sucesivos: mientras el segundo segmento hace las operaciones sobre los datos obtenidos en la primera etapa, el primer segmento ya realiza las operaciones con los datos que vienen con posterioridad a los anteriores. Siguiendo con el caso de la figura 36, una vez calculados los cuadrados de los datos en un periodo de reloj determinado, se pueden calcular otros nuevos con otros datos en el periodo siguiente y, al mismo tiempo, hacer la suma y la resta de los cuadrados que se habían calculado con anterioridad.

**Estructuras sistólicas**

Los esquemas de cálculo segmentados con almacenamientos diferenciados se denominan *estructuras sistólicas* porque el modo de hacer los cálculos se parece a la manera de funcionar del corazón: en cada sístole se expulsa un chorro de sangre. Traducido: en cada pulso de reloj se obtiene un dato resultante (de unos cálculos sobre unos datos que han entrado unos ciclos antes).

Figura 36. Esquema de cálculo con segmentación




En muchos casos, sin embargo, lo que interesa es reducir al máximo el número de registros adicionales y volver a utilizar los que contienen los datos de entrada, siempre que sea posible. (Esto suele imposibilitar que se puedan tomar nuevos datos de entrada a cada ciclo de reloj, tal como se ha comentado justo antes).

Siguiendo con el ejemplo, para reducir el número de registros, los valores de $x$ y de $x^2$ se deberían guardar en un mismo registro y, de manera similar, hacerlo con los de $y$ e $y^2$ y los de $r$ y $r^2$. Para que esto sea posible, los formatos de representación de los datos deben ser compatibles y, además, no han de interferir con otros cálculos que se puedan hacer en el sistema.

Como los retrasos en proporcionar la salida respecto al momento en el que cambia la entrada pueden ser muy diferentes según el recurso de cálculo de que se trate, la idea es que los segmentos introduzcan retrasos similares. En la segmentación del ejemplo, la etapa de la suma y de la resta se ha mantenido en un mismo segmento, suponiendo que las sumas son más rápidas que los productos. Hay que tener en cuenta que las multiplicaciones incluyen las sumas de los resultados parciales.

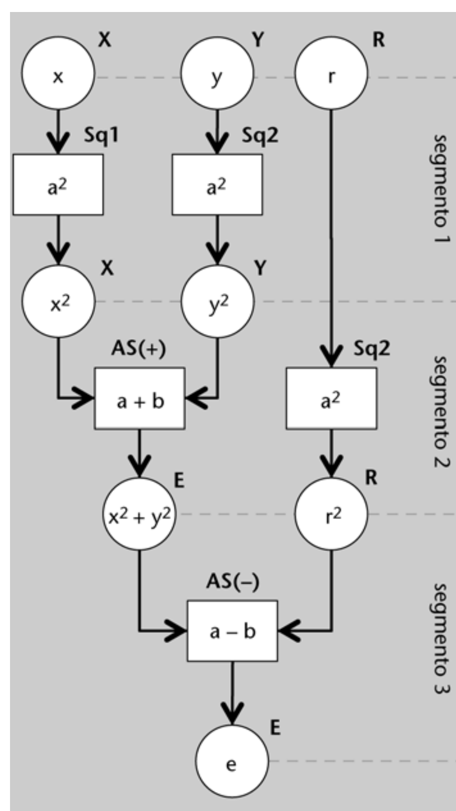
### 2.3. Esquemas de cálculo con recursos compartidos

La segmentación permite materializar esquemas de cálculo que trabajan con relojes de frecuencias más elevadas y compatibilizar los cálculos complejos con otros más simples en un mismo circuito.

El posible incremento en número de registros para almacenar datos intermedios puede ser compensado con la reducción de recursos de cálculo, ya que siempre los hay que no se utilizan en todos los segmentos. (Eso es cierto si no se utilizan estructuras sistólicas).

Así pues, la idea es utilizar una segmentación que permita utilizar un mismo recurso en distintas etapas del cálculo (o segmentos). Siguiendo con el ejemplo, los cuadrados de *r*, de *y* y de *x* pueden hacerse con el mismo recurso. Si, además, se introduce un nuevo segmento, la resta y la suma se podrían hacer con un mismo recurso, que fuera sumador/restador. El esquema de cálculo correspondiente se muestra en la figura 37.

Figura 37. Esquema de cálculo segmentado con recursos compartidos




El esquema de cálculo de la figura 37 ya está etiquetado. Es decir, cada nodo del grafo correspondiente tiene un nombre al lado que identifica el recurso al que está asociado. Por lo tanto, en un esquema de cálculo, las etiquetas indican la correspondencia entre nodos y recursos. En este caso, los recursos pueden ser de memoria (registros) o de cálculo.

Los recursos de cálculo que pueden realizar distintas funciones incorporan, en la etiqueta, la identificación de la función que tienen que hacer. Por ejemplo, “AS(–)” indica utilizar el módulo sumador/restador (“AS”, del inglés *adder/subtracter*) como restador. Se trata, pues, de recursos programables.

En extremo, se puede pensar en segmentar un esquema de cálculo de manera que utilice solo un único recurso de cálculo programable. Este único recurso de cálculo debería poder hacer todas las operaciones del esquema.

## 2.4. Materialización de esquemas de cálculo

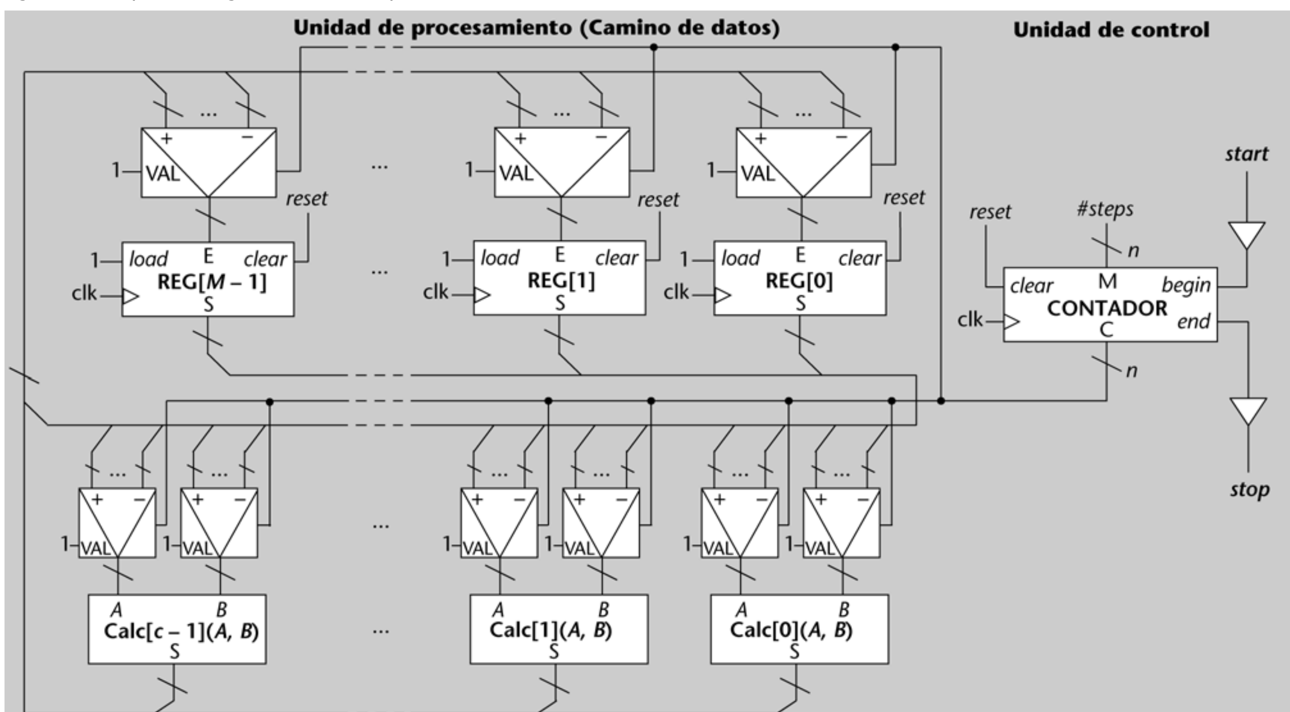
La materialización de un esquema de cálculo consiste en transformar su grafo etiquetado en el circuito correspondiente. Los circuitos se organizan de manera similar a como se ha visto para las máquinas de estados: se dividen en una unidad de procesamiento, que es el esquema de cálculo propiamente dicho, y una de control, que es un contador sencillo.

El modelo de construcción de la parte de procesamiento consiste en una serie de registros cuya entrada está determinada por un multiplexor que la selecciona según el segmento que se trate. La elección del segmento está hecha según un valor de control que coincide con el valor de salida de un contador. Si el contador es autónomo, entonces el procesamiento se repite cada cierto número de ciclos. Si es controlado externamente, como el de la figura 38, el procesamiento se hace cada vez que se recibe un pulso de inicio (*start*) y, al acabar, se pone el indicador correspondiente a 1 (*stop*). Hay que tener en cuenta que el contador deberá hacer la cuenta desde cero y hasta $M - 1$, donde $M$ es el número de pasos (#*steps*) o de segmentos del esquema de cálculo.

La organización de los recursos de cálculo sigue la misma que la de los de memoria (registros): las entradas quedan determinadas por multiplexores que las seleccionan según el segmento que se esté llevando a cabo del esquema de cálculo.

En la figura 38 se ha supuesto que todos los recursos trabajan con dos operandos para simplificar un poco el dibujo. Por el mismo motivo, no se muestran ni las entradas ni las salidas de datos. Habitualmente, las entradas de datos se hacen en el primer paso (paso número 0 o primer segmento) y las salidas se obtienen de un subconjunto de los registros en el último paso, en el mismo tiempo en que se pone *stop* a 1.

Figura 38. Arquitectura general de un esquema de cálculo

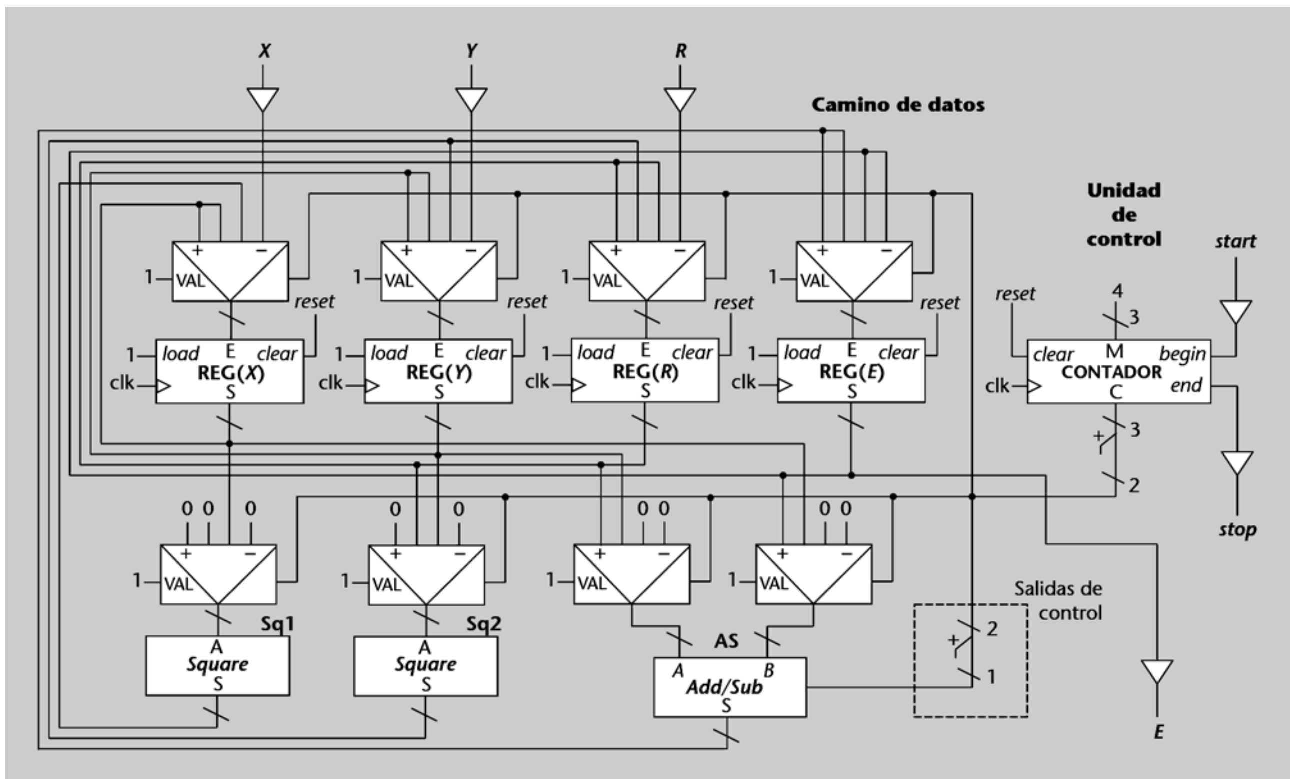


Hay dos buses de interconexión entre los recursos: el que recoge los datos de los recursos de cálculo y los lleva a los de memoria y el que sirve para la comunicación en sentido inverso. En la figura solo se ha marcado que cada elemento toma algunos bits del bus o los pone, sin especificar cuáles.

Para dejar más clara esta arquitectura, en la figura 39 podemos ver el circuito correspondiente al caso de ejemplo que se ha mostrado en la figura 37. Para construirlo, primero hay que poner multiplexores en las entradas de los registros y de los recursos de cálculo. Deben tener tantas entradas de datos como segmentos tiene el esquema de cálculo. Las entradas de los multiplexores de entrada a los registros que no se utilicen se deben conectar a las salidas de los mismos registros para que la carga no cambie su contenido. Las de los multiplexores para los recursos de cálculo que no se utilicen se pueden dejar a cero. Los buses de distribución de datos tienen que ponerse cerca: habrá tantos como registros para los multiplexores que seleccionan entradas para los elementos de cálculo, y tantos como recursos de cálculo para los de memoria. Las conexiones, entonces, son bastante directas observando el esquema de cálculo, segmento por segmento.

En el esquema del circuito se han dejado los multiplexores de 4 entradas para respetar la forma de construcción que se ha explicado. No obstante, de cara a su materialización, se podría simplificar considerablemente.

Figura 39. Circuito correspondiente a un esquema de cálculo para $x^2 + y^2 - r^2$




En este caso, el esquema de cálculo tiene dos salidas: *stop*, que indica que el cálculo se ha terminado, y *E*, que contiene el resultado del cálculo completo. De hecho, el registro *E* mantiene el valor del último cálculo hasta la etapa número

2, en la que se guarda $x^2 + y^2$. En el último segmento (etapa número 3) será cuando se calcule el valor final ($x^2 + y^2 - r^2$) y se almacene en $E$. Por lo tanto, al utilizar los esquemas de cálculo como submódulos de algún circuito mayor, hay que tener presente que el resultado solo será válido cuando *stop* = 1.

En este tipo de implementaciones, el estado se asocia directamente al camino que siguen los datos hasta obtener el resultado de este "recorrido": el valor del contador selecciona los operandos de cada operación y el registro de destino. No obstante, siempre puede haber un poco de lógica de control para calcular otras salidas, como la programación de algunos recursos de cálculo o la gestión de bancos de registros o memorias. En la figura 39 se puede ver un pequeño recuadro en el que se calcula la salida de control para el recurso programable de suma y resta.

**Camino de datos**

La denominación de **camino de datos** en la parte operacional es mucho más relevante en los esquemas de cálculo, ya que, de hecho, cada uno de sus segmentos equivale al camino que deben seguir los datos para obtener un resultado concreto.

**Actividades**

**10.** Diseñad el esquema de cálculo para:

$$a \cdot t^2/2 + v \cdot t + s$$

donde $a$ es la aceleración; $v$, la velocidad; $s$, el espacio inicial, y $t$, el tiempo.

Se puede suponer que se dispone de recursos de cálculo para sumar, multiplicar, dividir por 2 y hacer el cuadrado de un número. Todos los números tienen una anchura de 16 bits con un formato en coma fija y 8 bits para la parte fraccionaria.

Los **esquemas de cálculo** son representaciones de secuencias de operaciones encaminadas a obtener un determinado resultado. Según los criterios de diseño con los que se lleven a cabo, pueden necesitar más o menos recursos. Generalmente pueden resolverse de manera que el cálculo se haga lo más rápido posible, cueste lo que cueste, o con el menor coste posible, tarde lo que tarde. Los primeros esquemas serán muy paralelos, con muchos recursos de cálculo trabajando al mismo tiempo. Los otros, muy secuenciales, con el mínimo número de recursos posible.

## 2.5. Representación de máquinas algorítmicas

Los esquemas de cálculo son útiles para diseñar máquinas de cálculo, pero no sirven para representar cálculos condicionales o iterativos. Un cálculo condicional solo se efectúa si se cumplen unas determinadas condiciones. Uno iterativo necesita repetir una porción de las operaciones para obtener el resultado.

Si en un cálculo se introducen estas opciones, se transforma en un algoritmo, mientras que la máquina de cálculo correspondiente se trasforma en una **máquina algorítmica**.

La representación de los algoritmos se puede hacer de muchas maneras. Gráficamente se pueden utilizar **diagramas de flujo** con nodos de varios tipos:

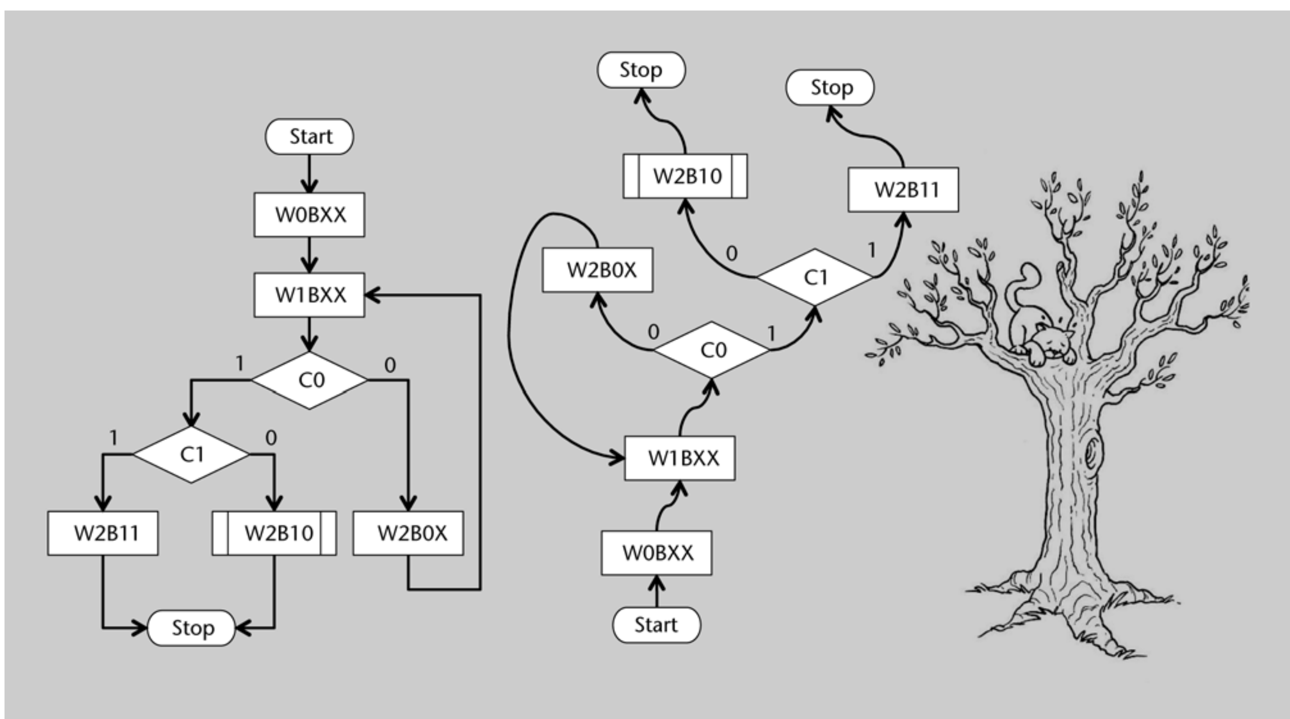
uno terminal, uno de procesamiento, uno de proceso predefinido (se explica qué es a continuación) y uno de decisión.

Como se muestra en la figura 40, los nodos terminales tienen la forma de cápsula y sirven para marcar el comienzo y el fin del algoritmo. Los de procesamiento y proceso predefinido se utilizan para representar las operaciones. Los dos son rectángulos, pero los últimos tienen una doble línea a cada lado. Las operaciones que se indican en los primeros se ejecutan simultáneamente. Los segundos se utilizan para los esquemas de cálculo. Los nodos de decisión son romboides. Se utilizan para introducir ramas de ejecución nuevas en función de si un bit de condición es uno (CIERTO) o cero (FALSO).

De hecho, un algoritmo se puede ver como una especie de árbol que tiene la raíz en un nodo terminal (el inicial) y un tronco del que nacen distintas ramas en cada nodo de decisión. La longitud de cada rama depende de los nodos de procesamiento/proceso predefinido que haya. Las hojas son siempre nodos terminales de finalización. (A diferencia de los árboles, algunas ramas pueden volver a unirse.)

En la figura 40 aparece una ilustración de un diagrama de flujo, con nodos de todos los tipos, y la analogía con los árboles.

Figura 40. Representación de una máquina algorítmica con un diagrama de flujo, en sentido de lectura a la izquierda y en forma de árbol a la derecha




Las representaciones de los diagramas de flujo son adecuadas para el diseño de algoritmos que finalmente se materialicen en forma de circuitos, ya que se focalizan en el procesamiento (cálculos en serie, condicionales e iterativos) y no en el control.

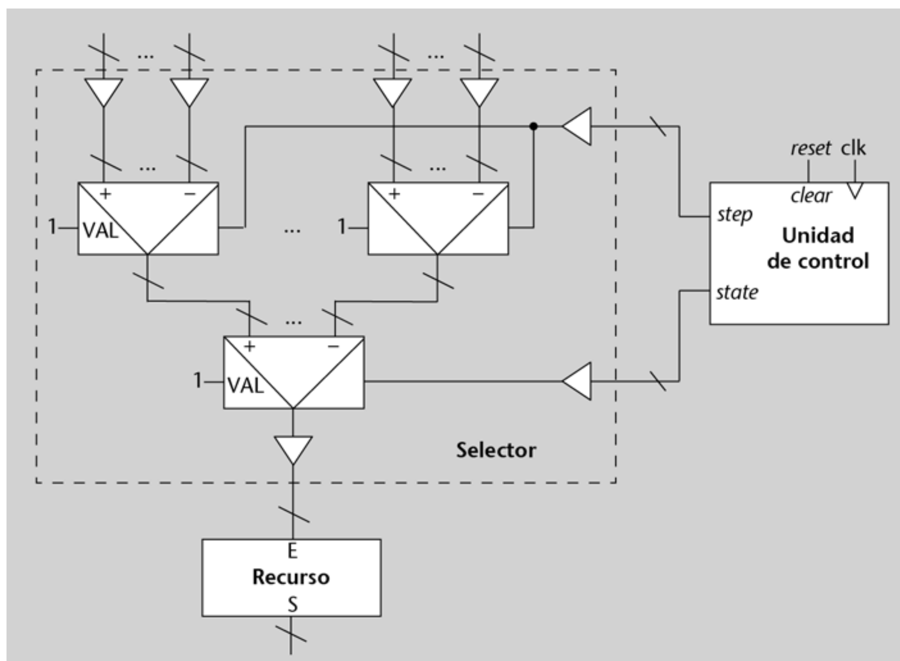
## 2.6. Materialización de máquinas algorítmicas

La implementación de los algoritmos sigue un esquema similar al de los esquemas de cálculo, con un peso más importante para la unidad de procesamiento que para la unidad de control, que es más sencilla.

Por lo tanto, la arquitectura de la unidad de proceso es igual a la que se ha presentado para los esquemas de cálculo, salvo por dos cosas: que hay recursos para calcular los bits de condición y que los multiplexores que seleccionan las entradas para los recursos son sustituidos por módulos “selectores”. Estos módulos tienen dos niveles multiplexores: el primer nivel es el que recibe la entrada del contador y el segundo el que selecciona la entrada según el estado.

En la figura 41 aparece un diagrama de bloques sobre cómo se organiza este módulo selector. Como ya sucedía con los esquemas de cálculo, estos módulos se pueden simplificar muchísimo en la mayoría de las implementaciones. Hay que tener en cuenta que hay muchos casos *don't-care* y entradas repetidas en los multiplexores.

Figura 41. Módulo de conexiones en la entrada de los recursos




La unidad de control de las máquinas algorítmicas también tiene un contador, que sirve para contar las etapas o los segmentos de los esquemas de cálculo correspondientes, si los hay. Pero también debe tener un registro para almacenar el estado en el que se encuentra (nodo de procesamiento o de proceso predefinido) y la lógica de cálculo del estado siguiente.

De hecho, se trata de una parte de control que se parece a la que se ha visto para las PSM, en la sección 1.3. Para utilizar el modelo de construcción que se presenta en dicha sección habría que transformar el diagrama de flujo en una

PSM. Esta transformación implica agrupar las distintas cajas de estado en secuencia en un único estado-programa y convertir los caminos que unen las cajas de estados en arcos de salida de las de origen con expresiones de condición formadas por el producto de las expresiones de las cajas de condición, complementadas o no, según si se trata de la salida 0 o de la 1 de cada caja.

También existe la opción de obtener una unidad de control directamente del diagrama de flujo, tal y como se verá a continuación. En este sentido, hay que tener presente la similitud entre los diagramas de flujo y las ASM, particularmente en el hecho de que los nodos de procesamiento o de proceso predefinido son equivalentes a las cajas de estado y los de decisión, a las de decisión.

Para evitar que tenga que haber una lógica de cálculo del estado siguiente, se puede optar por una codificación de tipo *one-hot bit*. En esta codificación, cada estado tiene asociado un biestable que se pone a 1 cuando la máquina de estados se encuentra, precisamente, en ese estado. (Se supone que, en un algoritmo orientado a hacer cálculos, el número de biestables que se utilizará será relativamente pequeño en comparación con la cantidad de registros de trabajo del circuito global.)

**Codificación *one-hot bit***

La codificación *one-hot bit* es la que destina un bit a codificar cada símbolo del conjunto que codifica. En otras palabras, codifica *n* símbolos en *n* bits y a cada símbolo le corresponde un código en el que solo el bit asociado está a 1, es decir, el símbolo número *i* se asocia al código con el bit en posición *i* a 1. Por ejemplo, si hay dos símbolos, *s*0 y *s*1, la codificación de bit único sería 01 y 10, respectivamente.

Con este tipo de unidades de control, las máquinas algorítmicas se encienden al recibir un pulso a 1 para la entrada *start* y generan un pulso a 1 para la salida *stop* en el último ciclo de reloj en el que se ejecuta el algoritmo asociado. En otras palabras, la idea es que reciben un 1 por la entrada *start*, y que este 1 se va desplazando por los biestables de la unidad de control correspondiente hasta llegar a la salida. De hecho, cada desplazamiento es un estado diferente. Cada biestable representa, por lo tanto, un estado o, si se quiere, un nodo de procesamiento o de proceso definido.

En la figura 42 aparece el circuito de la unidad de control del diagrama de flujo que se ha visto en la figura 40. En el esquema, los arcos están marcados con buses de línea gruesa para facilitar la comparación entre circuito y diagrama. Además, los nodos de estado se identifican con el mismo nombre con el que aparecen en el diagrama de flujo correspondiente. Si estos nodos tienen más de una entrada, se debe poner una puerta OR que las una, tal y como se puede observar en el circuito con la puerta OR1. La puerta OR1 se ocupa de "dejar pasar" el 1, que puede llegar del nodo de procesamiento inicial (W0BXX) o de un nodo posterior (W2B0X).

Los nodos de decisión son elementos que operan igual que un demultiplexor: transmiten la entrada a la salida seleccionada. En la figura aparecen dos: DMUX0 y DMUX1. Por ejemplo, DMUX0 "hace pasar" el 1 que viene de W1BXX hacia W2B0X, o hacia el otro nodo de decisión según la condición *C*0.

Los nodos que se corresponden con un proceso predefinido (con un esquema de cálculo, por ejemplo) activan un contador. Este contador es compartido por todos los nodos de proceso predefinido, ya que solo puede haber uno activo al mismo tiempo. En el ejemplo de la figura 40 vemos uno, que es W2B10.

Como la máquina de control debe estar en W2B10 mientras el contador no llegue al máximo, el biestable correspondiente se pone a 1 tanto por el arco de entrada que viene del nodo de decisión de *C*1 como por el hecho de que se haya activado y la cuenta no haya terminado (AND1).

En el caso general, la entrada que activa el contador (*inc*) está formada por la suma lógica de todas las señales de los biestables correspondientes a los nodos de proceso predefinido. Y la entrada *A* del comparador está proporcionada por un multiplexor que selecciona los diferentes valores de $M - 1$ (donde *M* representa el número de ciclos que hay que contar) según el estado. Sin embargo, en este caso no es necesaria ni la puerta OR ni el multiplexor porque solo hay un proceso predefinido, que es W2B10.

El arco de salida de los nodos que se corresponden con procesos predefinidos es la salida de una puerta (AND2), que comprueba que el estado está en activo y la cuenta ya ha terminado.

Figura 42. Una arquitectura de la unidad de control de una máquina algorítmica

La salida *stop* está formada por la suma lógica de todos los arcos que conducen al nodo terminal de finalización.

Es importante observar que esta unidad tiene, como señales de entrada, *start* y los bits de condición de la unidad operacional, *C*0 y *C*1. Y, como señales de salida, *stop* y los códigos de estado (*state*) y de segmento (*step*) para gobernar los selectores de la unidad operacional.

Si el número de nodos de un diagrama de flujo es muy alto, la unidad de control también lo será. Como se verá más adelante, una posible solución para la implementación de máquinas algorítmicas complejas es el uso de memorias ROM que almacenen las tablas de verdad para las funciones de cálculo del estado siguiente.

### 2.7. Caso de estudio

A la hora de diseñar una máquina algorítmica se debe partir de un diagrama de flujo con el mínimo número de nodos posible: cuantos menos nodos, más pequeña puede ser la unidad de procesamiento y más simple puede ser la unidad de control.

El caso de estudio que se presentará es el de construir un multiplicador secuencial. La idea es reproducir el mismo procedimiento que se lleva a cabo con las multiplicaciones a mano: se hace un producto del multiplicando por cada dígito del multiplicador y se suman los resultados parciales de manera decalada, según la posición del dígito del multiplicador. En binario, las cosas son más sencillas, ya que el producto del multiplicando por el bit del multiplicador que toque se convierte en cero o en el mismo multiplicando. Así, una multiplicación de números binarios se convierte con una suma de multiplicandos decalados según la posición de cada 1 del multiplicador.

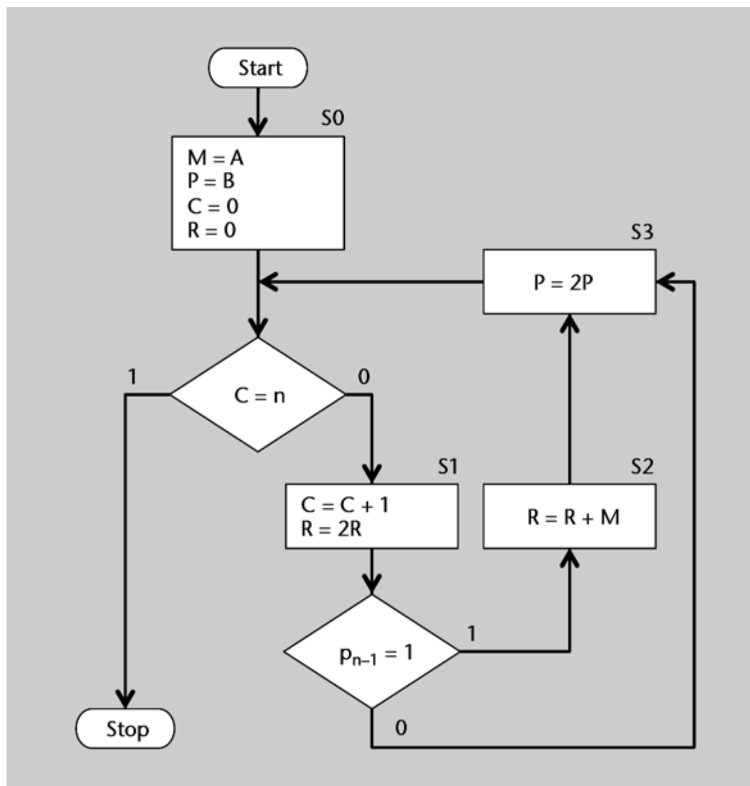
Para entenderlo, a continuación se realiza el producto de 1011 (multiplicando) por 1010 (multiplicador):

| | | | | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| | | | x | 1 | 0 | 1 | 0 |
| | | | | 0 | 0 | 0 | 0 |
| | | | 1 | 0 | 1 | 1 | |
| | | 0 | 0 | 0 | 0 | | |
| | 1 | 0 | 1 | 1 | | | |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Para materializar un circuito que lleve a cabo la multiplicación con este procedimiento, es necesario, en primer lugar, obtener un diagrama de flujo de las operaciones y de las decisiones que se toman en él.

En la figura 43 encontramos el algoritmo correspondiente: el multiplicando es *A*; el multiplicador, *B*, y el resultado es *R*. Internamente, *A* y *B* se almacenan en las variables *M* y *P*, respectivamente. El indicador de la posición del dígito de *B* con el que se debe hacer un determinado producto se obtiene de un contador, *C*. Además, el contador sirve de control del número de iteraciones del algoritmo, que es el número de bits de los operandos *A* y *B* (*n*).

Figura 43. Diagrama de flujo de un multiplicador binario




En los diagramas de flujo para las máquinas algorítmicas, los diferentes cálculos con variables no se expresan de la forma:

$$V^{+} = \text{expresión}$$

sino de la forma:

$$V = \text{expresión}$$

porque son representaciones de algoritmos que trabajan, mayoritariamente, con datos almacenados en variables. No obstante, hay que tener presente que esto se hace para simplificar la representación: las variables actualizan su valor al finalizar el estado en el que se calculan. Se supondrá que no hay señales de salida que no sean variables.

Hay que señalar que el resultado *R* deberá tener una amplitud de 2*n* bits y que al contador le basta con el número entero más próximo por la derecha a $\log_2 n$, que se identificará en el circuito con *m*, es decir, que el contador será de *m* bits, siendo *m* lo suficientemente grande como para representar cualquier número más pequeño que *n*.

El algoritmo de multiplicación empieza (*S*0) cargando el multiplicando en *M* y el multiplicador en *P*, así como inicializando las variables de trabajo. En este caso, *R* será también la que contendrá el resultado de salida.

Dado que debe hacer tantos productos como bits tenga el multiplicador *P*, hay un nodo de decisión que sirve para controlar el número de iteraciones: cuando *C* sea *n* ya se habrán hecho todos los productos parciales y el resultado estará en *R*.

En cada iteración se incrementa el contador *C* y se desplaza el resultado previo hacia la izquierda (*S*1). Esto último se justifica porque el resultado final se obtiene haciendo las sumas parciales de los productos del multiplicando *M* por los bits del multiplicador de izquierda a derecha. Así, antes de sumar a *R* un nuevo producto, hay que multiplicarlo por dos porque el resultado parcial que contiene es la suma parcial previa, que se corresponde con los productos de *M* por los bits más significativos de *P*.

El proceso en *S*2 consiste, simplemente, en hacer la suma del producto de un bit a 1 de *P* por *M*, con el resultado parcial previo en *R*. Con independencia de este proceso, en *S*3 se hace un desplazamiento a la izquierda de *P* para obtener el siguiente multiplicador.

Los desplazamientos de *P* y *R* ahorran el uso de multiplexores o de otro tipo de selectores, ya que los bits con los que se debe trabajar siempre están en posiciones fijas.

Para entenderlo, a continuación podemos ver la multiplicación del ejemplo anterior, representada de acuerdo con las operaciones del algoritmo que se ha presentado. (Los desplazamientos de *P* no se muestran, pero se pueden deducir del subíndice del bit que se utiliza para el producto, que aquí sí que cambia.)

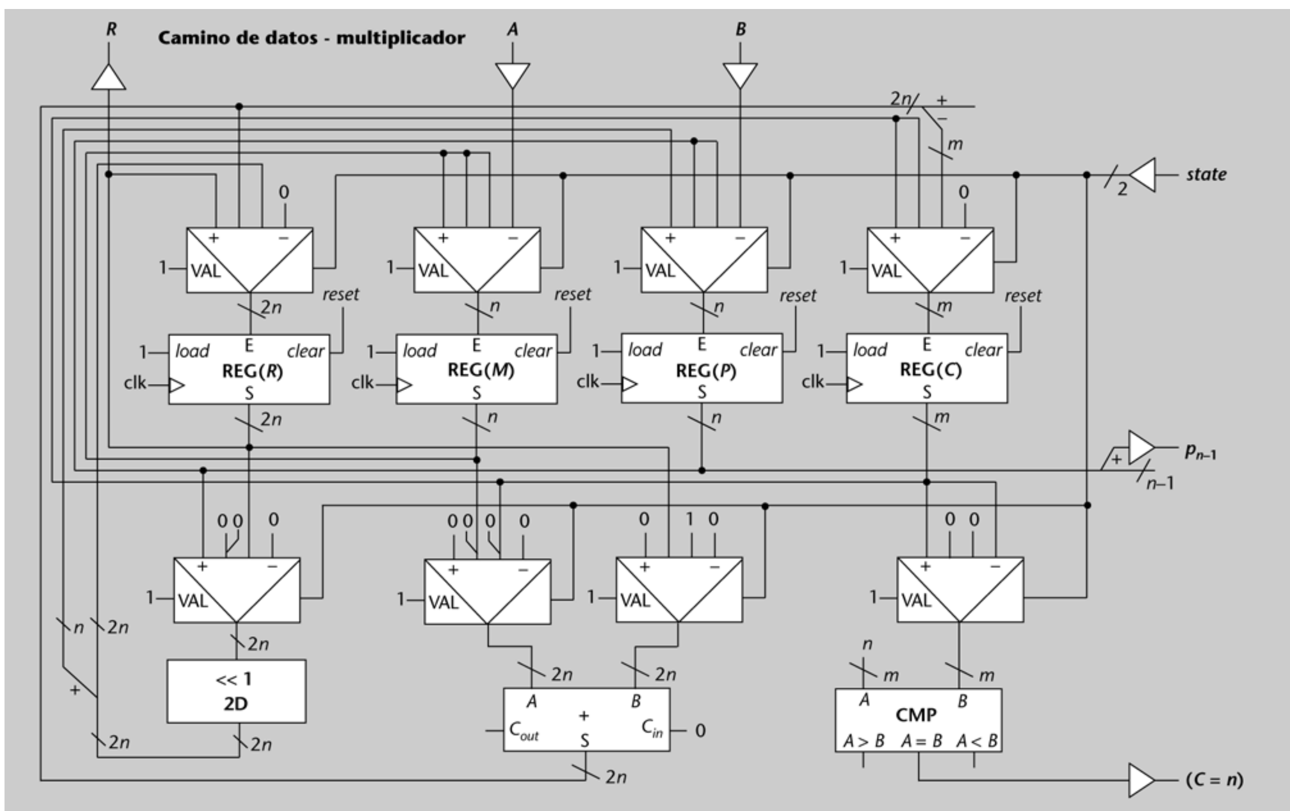
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 0 | 1 | 1 | $M$ |
| | | | x | 1 | 0 | 1 | 0 | $P$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $R = 2 \cdot R$ |
| | | | + | 1 | 0 | 1 | 1 | $M \cdot p_{n-1}$ |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | $R = R + M \cdot p_{n-1}$ |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | $R = 2 \cdot R$ |
| | | | + | 0 | 0 | 0 | 0 | $M \cdot p_{n-2}$ |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | $R = R + M \cdot p_{n-2}$ |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | $R = 2 \cdot R$ |
| | | | + | 1 | 0 | 1 | 1 | $M \cdot p_{n-3}$ |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | $R = R + M \cdot p_{n-3}$ |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | $R = 2 \cdot R$ |
| | | | + | 0 | 0 | 0 | 0 | $M \cdot p_{n-4}$ |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | $R = R + M \cdot p_{n-4}$ |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | |

La unidad de procesamiento seguirá la arquitectura que se ha presentado, pero sin necesidad de selectores de dos niveles, ya que no hay procesos predefinidos: todo son procesamientos que se pueden realizar en paralelo. Esto también simplificará la unidad de control.

Para construir la unidad de procesamiento, en primer lugar hay que tener claros los recursos de memoria y de cálculo con los que debe contar. En cuanto a los primeros, basta con un registro para cada variable del algoritmo: $M$ y $P$ de $n$ bits, $R$ de $2n$ bits y $C$ de $m$ bits. En cuanto a los segundos, debe haber uno por operación diferente, como mínimo. En este caso, tanto las sumas como los decaladores pueden ser compartidos porque se utilizan en estados diferentes, hecho que es positivo, dado que se reduce el número de recursos de cálculo. En este caso, solo es necesario un sumador para calcular $C + 1$ y $R + M$ porque son sumas que se realizan en estados diferentes ($S1$ y $S2$, respectivamente) y solo es necesario un decalador que se puede aprovechar para realizar el cálculo de $2R$ en $S1$ y $2P$ en $S3$.

El hecho de que el decalador y el sumador sean compartidos implica que deben trabajar con datos de tantos bits como el máximo de bits de los operandos que puedan recibir. En este caso, como $R$ es de $2n$ bits, hay que ajustar el resto de los operandos a esta amplitud: en el caso del decalador, se añaden $n$ bits a 0 a la derecha de $P$, en la entrada más significativa, para llegar a $2n$ bits, y en el caso del sumador, los operandos $C$ y $M$, en las entradas 1 y 2 del multiplexor de la izquierda, se pasan a $2n$ bits, a los que se añaden 0 por la izquierda, para no cambiar su valor. (En la figura no aparecen las amplitudes de los buses para mantener la legibilidad.)

Figura 44. Unidad de procesamiento de un multiplicador en serie

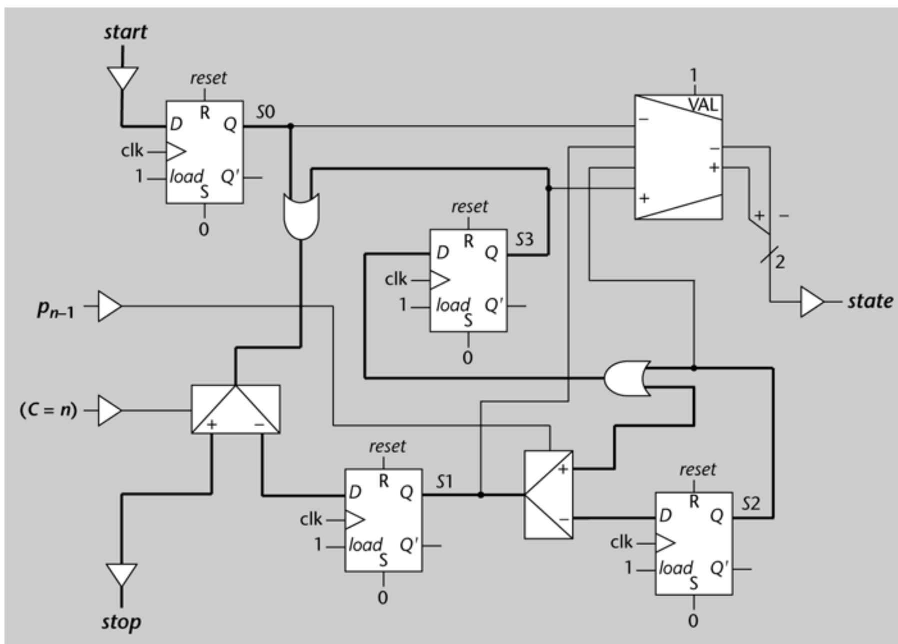


La unidad de procesamiento mostrada en la figura 44 sigue fielmente el modelo de construcción, pero se puede simplificar con respecto al uso de los multiplexores.

La unidad de control sigue el modelo de construcción presentado en el apartado anterior, con una codificación de los estados de tipo *one-hot bit* que permite el traslado directo de la estructura del diagrama de flujo a la del circuito correspondiente. Sin embargo, tiene un aspecto diferente del visto en la figura 42 porque no hay ningún contador. Las entradas al codificador del estado deben seguir el mismo orden que las entradas de los multiplexores de los selectores de la unidad de procesamiento.

En la figura 45 aparece el esquema del circuito de control correspondiente al multiplicador. Las líneas de conexión que se corresponden con los arcos del diagrama de flujo correspondiente se destacan en negrita.

Figura 45. Unidad de control de un multiplicador en serie




En este caso, dado que el número de estados (4) y de entradas (2) es pequeño, también se puede construir una unidad de control a partir de la tabla de transiciones. Para hacerlo, es conveniente ver el diagrama de flujo como un grafo de estados en el que los nodos de procesamiento son estados y los nodos de decisión condiciones de las transiciones, de manera similar a como se trata en las ASM.

### Actividad

**11.** Rehaced los esquemas de los circuitos de la unidad de procesamiento y de la de control, si conviene, para reducir al máximo el número de multiplexores y la cantidad de entradas de cada uno de ellos en la unidad de procesamiento.

# 3. Arquitectura básica de un computador

Un computador es una máquina capaz de procesar información. Según esta definición, en realidad estamos en un mundo lleno de computadores, algunos de los cuales vemos como tales: ordenadores portátiles y de sobremesa. Pero la gran mayoría no los percibimos como ordenadores porque se encuentran empotrados (y ocultos) en otros objetos: dispositivos móviles, sistemas de cine en casa, electrodomésticos, coches, aviones, etc. Según la variabilidad de los algoritmos que pueden ejecutar para procesar la información, se puede ir desde los más sencillos, constituidos por máquinas algorítmicas especializadas, hasta los más complejos.

En este apartado se tratará sobre cómo están construidos los computadores y cómo funcionan, y empezaremos el recorrido por la transformación de las máquinas algorítmicas en computadores para acabar en las arquitecturas de los computadores más completos.

En primer lugar se verán las máquinas algorítmicas que son capaces de interpretar secuencias de códigos de acciones y operaciones, es decir, que pueden ejecutar programas diferentes cambiando exclusivamente el contenido de la memoria en la que se almacenan. De hecho, este tipo de máquinas se denominan **procesadores** porque "procesan" datos según un programa determinado.

A medida que aumenta el repertorio de funciones que pueden realizar los programas, es decir, el conjunto de instrucciones diferentes que incluyen, las máquinas algorítmicas que las pueden entender también se vuelven más complejas. En el segundo subapartado se verá cómo construir un procesador capaz de ejecutar programas relativamente complejos y que se organiza en dos grandes bloques: el de memoria y el de procesamiento. Se explicará cómo se interpretan las instrucciones y la relación entre estos dos bloques.

El tercer subapartado se dedica a explicar arquitecturas de procesadores (**microarquitecturas**) más complejas que la del procesador sencillo y cómo se logra que puedan ejecutar más instrucciones por unidad de tiempo. También se explicarán los diferentes tipos de procesadores que puede haber según las características que tienen.

Como los procesadores se ocupan de procesar información pero no de obtenerla ni de proporcionarla, deben ir acompañados de otros componentes que realicen estas funciones, es decir, que se ocupen de la entrada y salida de la información. Este conjunto recibe el nombre de **computador**. El cuarto y último subapartado explica cómo están construidos los computadores tanto en general como los que se orientan a aplicaciones específicas.

### 3.1. Máquinas algorítmicas generalizables

Las máquinas algorítmicas se construyen para ejecutar un único algoritmo de la manera más eficiente posible. Es decir, llevan a cabo la funcionalidad indicada en el algoritmo minimizando su coste. En principio, el coste de ejecución se determina según factores como el tiempo, el consumo de energía y el tamaño de los circuitos.

La evolución de la microelectrónica ha permitido que, en un mismo espacio, cada vez se puedan poner circuitos más complejos. Por lo tanto, cada vez se pueden construir máquinas con una funcionalidad mayor. Pero también es necesario que los sistemas resultantes tengan un coste razonable respecto al rendimiento que se obtiene de ellos. Por lo tanto, la batalla por la eficiencia está en el tiempo y el consumo de energía: se quieren obtener resultados cada vez más rápidamente y consumiendo poca energía.

Por ejemplo, se quiere que los "teléfonos móviles inteligentes" permitan ver películas de alta calidad con baterías de larga duración y poco voluminosas que faciliten la portabilidad de los dispositivos.

**Funcionalidad**

La **funcionalidad**, en este contexto, hace referencia al conjunto de funciones que puede llevar a cabo un sistema. La funcionalidad de un algoritmo está formada por todas aquellas funciones que se pueden diferenciar de manera externa. Por ejemplo, la funcionalidad de un algoritmo de control de un *gadget* de audio portátil incluye funciones para hacer sonar un tema, dejarlo en pausa, pasar al siguiente, etc.

Otra parte del coste está relacionada con el desarrollo y posterior mantenimiento de los productos gobernados por máquinas algorítmicas: cuanto más compleja es la funcionalidad de un sistema, más difícil es implementar la máquina correspondiente. Además, cualquier cambio en la funcionalidad (es decir, en el algoritmo) implica construir una máquina algorítmica nueva.

Con el estado actual de la tecnología, lo que más pesa a la hora de materializar un producto es la funcionalidad (porque incrementa su "valor añadido"), la facilidad de desarrollo (porque reduce "el tiempo de puesta en el mercado") y el mantenimiento posterior (porque facilita su actualización). De hecho, la tecnología permite ejecutar de manera eficiente gran parte de la funcionalidad genérica de los algoritmos. Solo unas pocas funciones son realmente críticas y hay que llevarlas a cabo con algoritmos específicos.

**Función crítica**

Se dice que una función es crítica cuando tiene un peso muy elevado en el coste de ejecución del algoritmo que la incluye. Por ejemplo, porque las demás funciones dependen de ella o porque tiene mucha influencia en los parámetros que determinan el coste o la eficiencia final del algoritmo global.

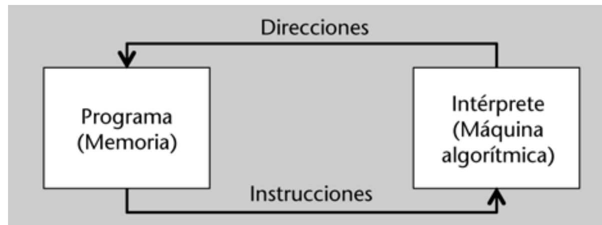
Así pues, es conveniente utilizar máquinas más genéricas, que puedan ejecutar conjuntos de funciones diferentes según la aplicación a la que se destinen o la propia evolución del producto en el que se encuentren. Para llevarlo a cabo, los algoritmos deberían estar almacenados en módulos de memoria, de manera que las máquinas llevarían a cabo los algoritmos que, en un momento determinado, tuvieran guardados.

Una **máquina algorítmica generalizable** sería aquella capaz de interpretar las instrucciones del programa (los pasos del algoritmo) que hubiera en la memoria correspondiente.

La máquina algorítmica que interpretara las instrucciones almacenadas en la memoria correspondiente sería, de hecho, la unidad encargada de hacer el procesamiento del programa en memoria. Para poder construir esta máquina,

es necesario determinar qué instrucciones debe interpretar y cómo se codifican en binario. A modo de ejemplo, en el apartado siguiente se describe una máquina de este tipo.

Figura 46. Esquema de la relación entre la memoria de programa y la máquina algorítmica de interpretación




### 3.1.1. Ejemplo de máquina algorítmica general

Las máquinas algorítmicas que interpretan programas almacenados en memoria son genéricas porque son capaces de ejecutar cualquier programa, aunque no lo pueden hacer con la eficiencia de las máquinas específicas, evidentemente. Por lo tanto, se convierten en máquinas de procesamiento de programas o **procesadores**. En este apartado veremos uno muy pequeño: el Femtoproc.

La idea es detallar cómo se puede construir uno de estos intérpretes aprovechando lo que se ha visto en las secciones anteriores e ilustrar así el funcionamiento interno de los procesadores.

El Femtoproc es una pequeña máquina algorítmica que interpreta un repertorio mínimo de instrucciones: una para sumar (`ADD`), dos para realizar operaciones lógicas bit a bit (`NOT` y `AND`) y, finalmente, una de salto condicional (`JZ`), que efectuará un salto en la secuencia de instrucciones a ejecutar si la última operación ha dado como resultado un 0.

A modo de curiosidad, este repertorio permite hacer cualquier programa, por sofisticado que sea. (El número de instrucciones, sin embargo, podría ser enorme, eso sí.) Hay que tener en cuenta que la suma también permite hacer restas, si se trabaja con números en complemento a 2, que con NOT y AND se puede construir cualquier función lógica y que el salto condicional permite alterar la secuencia de instrucciones para tomar decisiones y se puede utilizar como salto incondicional (forzando la condición) para hacer iteraciones.

Las instrucciones se codifican con 8 bits (1 byte) según el formato siguiente:

| Instrucción | Bits | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | 0 | 0 | operando$_1$ | | | operando$_0$ | | |
| AND | 0 | 1 | operando$_1$ | | | operando$_0$ | | |
| NOT | 1 | 0 | operando$_1$ | | | operando$_0$ | | |
| JZ | 1 | 1 | dirección de salto | | | | | |

Los operandos de las operaciones ADD, AND y NOT se obtienen de un banco de registros que, dada la codificación (se utilizan 3 bits para identificar los operandos), debe tener 8 registros: *R*7, *R*6, *R*5, *R*4, *R*3, *R*2, *R*1 y *R*0. Todos los registros son de 8 bits.

Para facilitar la programación, *R*0 y *R*1 son constantes, es decir, son registros en los que no se puede escribir nada. (De hecho, son falsos registros.) *R*1 tiene el valor 01h (unidad), de manera que sea posible construir cualquier otro número y, especialmente, que se puedan hacer incrementos y decrementos. *R*0, en cambio, tiene el valor 80h = $10000000_{(2}$ para poder averiguar el signo de los números enteros.

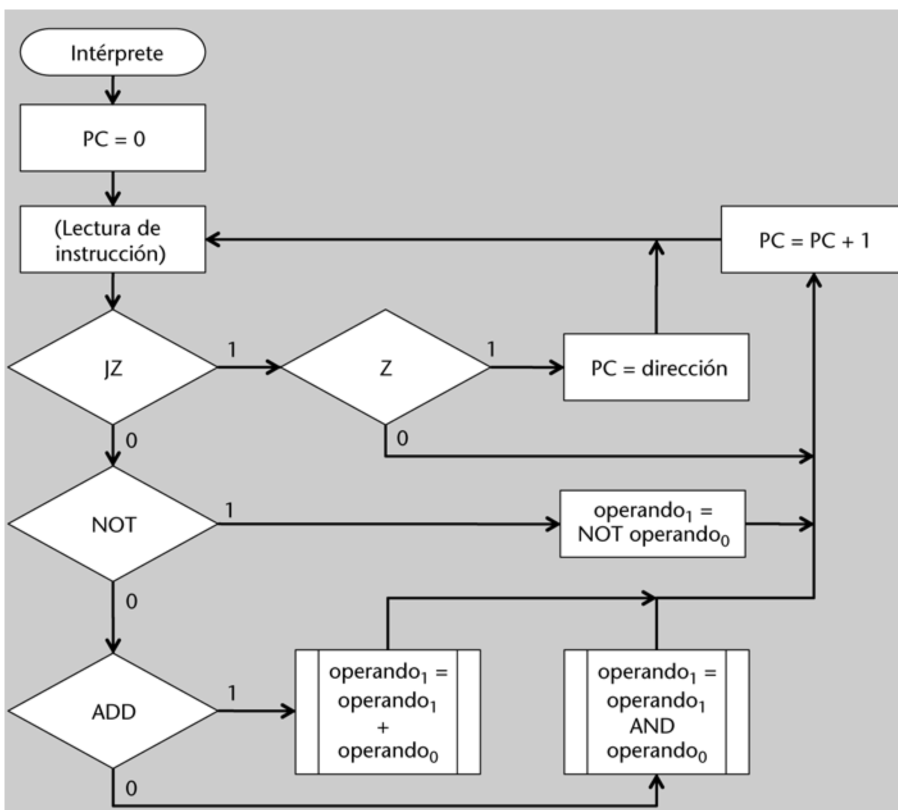
El resultado de las operaciones se almacenará en el $operando_1$. Si es *R*0 o *R*1, el resultado se perderá, pero el bit que identifica si la última operación ha sido cero o no (*Z*) sí que se actualizará. *Z* identifica un biestable que almacena esta condición.

Como la instrucción `JZ` puede expresar direcciones de 6 bits, los programas se almacenarán en una ROM de $2^6$ = 64 posiciones, de 8 bits cada una.

Con esta información, ya es posible realizar el algoritmo de interpretación de instrucciones y materializarlo con la máquina algorítmica correspondiente.

En este algoritmo se utiliza una variable que hace de contador de programa o *program counter* (PC), en inglés. Se trata de una variable que contiene la posición de memoria de la instrucción que se debe ejecutar. Como se ve en el diagrama de flujo de la figura 47, el algoritmo consiste en un "bucle infinito" que empieza por la lectura de la instrucción número 1, que se encuentra en la dirección 0.

Figura 47. Diagrama de flujo de la máquina de interpretación de {`ADD`, `AND`, `NOT`, `JZ`}



El diagrama de flujo anterior se puede detallar un poco más, conociendo la codificación de las instrucciones que interpreta el Femtoproc. En la tabla siguiente se puede ver la equivalencia entre las descripciones de los nodos del diagrama de flujo y las operaciones que están vinculadas al mismo.

| Proceso o condición | Operaciones | Efecto |
|---|---|---|
| (Lectura de instrucción) | $Q = M[PC]$ | Obtención del código de la instrucción que hay en la posición *PC* de memoria |
| ¿Salto? | $q_7 \cdot q_6$ | Cálculo del bit que indica si la instrucción es JZ |
| ¿Cero? | $z'$ | Indicación de que la última operación ha tenido, como resultado, el valor 0 |
| *PC* = dirección | $PC = Q_{5-0}$ | Asignación del valor de la dirección de salto a *PC* |
| *PC* = *PC* + 1 | $PC = PC + 1$ | Asignación del valor de *PC* + 1 a *PC* |
| ¿NOT? | $q_7 \cdot q'_6$ | Cálculo del bit que indica si la instrucción es NOT |
| $operando_1$ = NOT $operando_0$ | BR[$Q_{5-3}$] = NOT BR[$Q_{2-0}$] | Complemento de los bits del registro en posición $Q_{2-0}$ del banco de registros (BR) y asignación al registro en posición $Q_{5-3}$ |
| ¿Suma? | $q'_7 \cdot q'_6$ | Determinación de que la instrucción sea ADD |
| $operando_1 = operando_1 + operando_0$ | 0: $B$ = BR[$Q_{2-0}$];<br>1: BR[$Q_{5-3}$] = BR[$Q_{5-3}$] + $B$; | Ejecución del esquema de cálculo que obtiene la suma de los registros $Q_{2-0}$ y $Q_{5-3}$ del BR y asignación al registro en posición $Q_{5-3}$ |
| $operando_1$ =<br>$operando_1$ AND $operando_0$ | 0: $B$ = BR[$Q_{2-0}$];<br>1: BR[$Q_{5-3}$] = BR[$Q_{5-3}$] AND $B$; | Ejecución del esquema de cálculo que obtiene la AND entre los bits de los registros $Q_{2-0}$ y $Q_{5-3}$ del BR y asignación al registro en posición $Q_{5-3}$ |

Como se puede observar en la tabla anterior, el nodo de procesamiento correspondiente a la lectura de la instrucción no es necesario, ya que el código de operación de la instrucción, *Q*, se obtiene directamente de la salida de datos de la memoria del programa. Como se verá, en máquinas más complejas la codificación de las instrucciones obliga a almacenar el código en una variable para ir descodificándolas de manera progresiva.

También hay dos esquemas de cálculo, uno para la suma y otro para el producto lógico (AND), que deben estar forzosamente segmentados si se utiliza un banco de registros con una entrada y una única salida de datos, ya que es necesario disponer de un registro auxiliar (*B*) que almacene temporalmente uno de los operandos.

La materialización de la máquina algorítmica correspondiente se muestra a continuación solo a modo de ejemplo. Como se verá, el diseño de estas máquinas es más un arte que una ciencia, ya que hay que transformar los diagramas de flujo correspondientes hasta llegar a los de partida para la implementación.

**Implementación del Femtoproc**

Si se partiera del diagrama de flujo anterior, se obtendría una máquina que funcionaría correctamente, pero sería ineficiente. En concreto, si se asignara un estado por nodo de procesamiento o nodo de proceso predefinido (los esquemas de cálculo correspondientes necesitan, obligatoriamente, dos estados de cuenta), la máquina tendría 8 estados: "PC = 0", "PC = dirección", "PC = PC + 1", "NOT", "ADD" (x2) y "AND" (x2).

Simplificando estados, el estado inicial "PC = 0" solo debe poner el registro *PC* a 0, lo que ya se hace con el *reset* inicial. Por lo tanto, se puede suprimir. Ya son 7.

Si observamos las operaciones de los esquemas de cálculo, comprobamos que la primera operación es igual. Por lo tanto, se pueden separar en dos nodos de procesamiento y hacer que el primero sea compartido. Ya solo quedan 6.

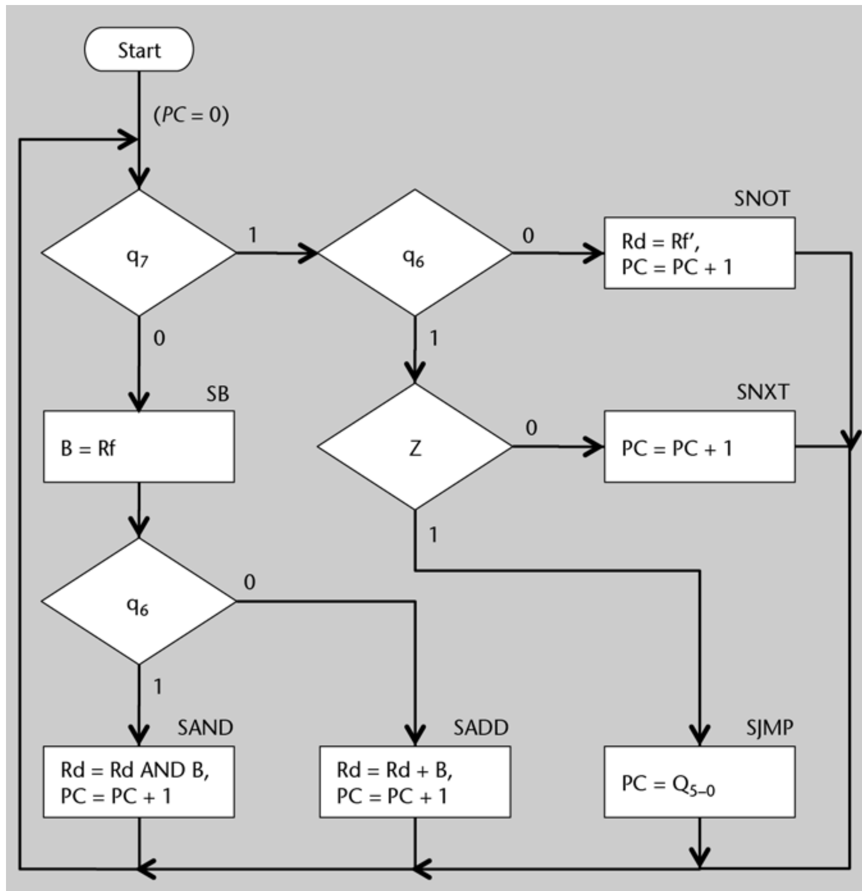
En la figura 48 se puede comprobar esta simplificación: se ha reducido el número de nodos y minimizado los esquemas de cálculo. En la figura se utiliza Rd (registro de destino) para representar el registro en posición $Q_{5-3}$ del banco de registros (BR), que sustituye el símbolo $operando_1$, y Rf (registro fuente) por BR[$Q_{2-0}$]. Los nodos de procesamiento llevan en la parte derecha superior el nombre que identifica el estado correspondiente.

Los nodos de decisión se han alterado para tomarse de acuerdo con los bits del código de instrucción. Así, para determinar si es JZ la expresión es $q_7 \cdot q_6$, y para NOT, $q_7 \cdot q'_6$, lo que se puede transformar en averiguar primero si $q_7 = 1$ y, entonces, según $q_6$, saber si es JZ o NOT. Algo similar se hace para discernir entre ADD y AND, que tienen un punto en común: que el valor del primer operando se debe almacenar en el registro *B*. El segundo paso sí que es diferente.

El camino de datos necesita los recursos que se enumeran a continuación.

- **De memoria**: uno para el contador de programa (*PC*), un biestable para almacenar la indicación de si la última operación ha sido cero o no (*Z*), un registro auxiliar para uno de los operandos (*B*) y un banco de registros (BR) con 8 registros de 8 bits, de los que los dos primeros poseen valores fijados.
- **De cálculo**: dos sumadores (uno para incrementar el *PC* y otro para la suma de datos), un producto lógico de buses, un complementador y la memoria ROM que contiene el programa (se podría decir que es el circuito que relaciona el valor del *PC* con el de la instrucción que se debe ejecutar, $Q = M[PC]$).
- **De conexión**: buses, un multiplexor de selección del próximo valor para el *PC*, que puede ser $PC + 1$ o $Q_{5-0}$, otro del resultado que se ha de cargar en el BR y otro para elegir qué registro se quiere leer, si Rd o Rf.

Figura 48. Diagrama de flujo del Femtoproc adaptado para la materialización




La codificación de los estados será de tipo *one-hot bit*, de manera que la unidad de control se pueda implementar directamente a partir del diagrama de flujo, tal y como se ha visto.

La tabla que relaciona los estados con las operaciones del camino de datos es la siguiente. Todas las señales denominadas con "ld_" se conectan a las entradas de carga de contenido de los registros asociados. Los controles de los multiplexores son señales que se deno-

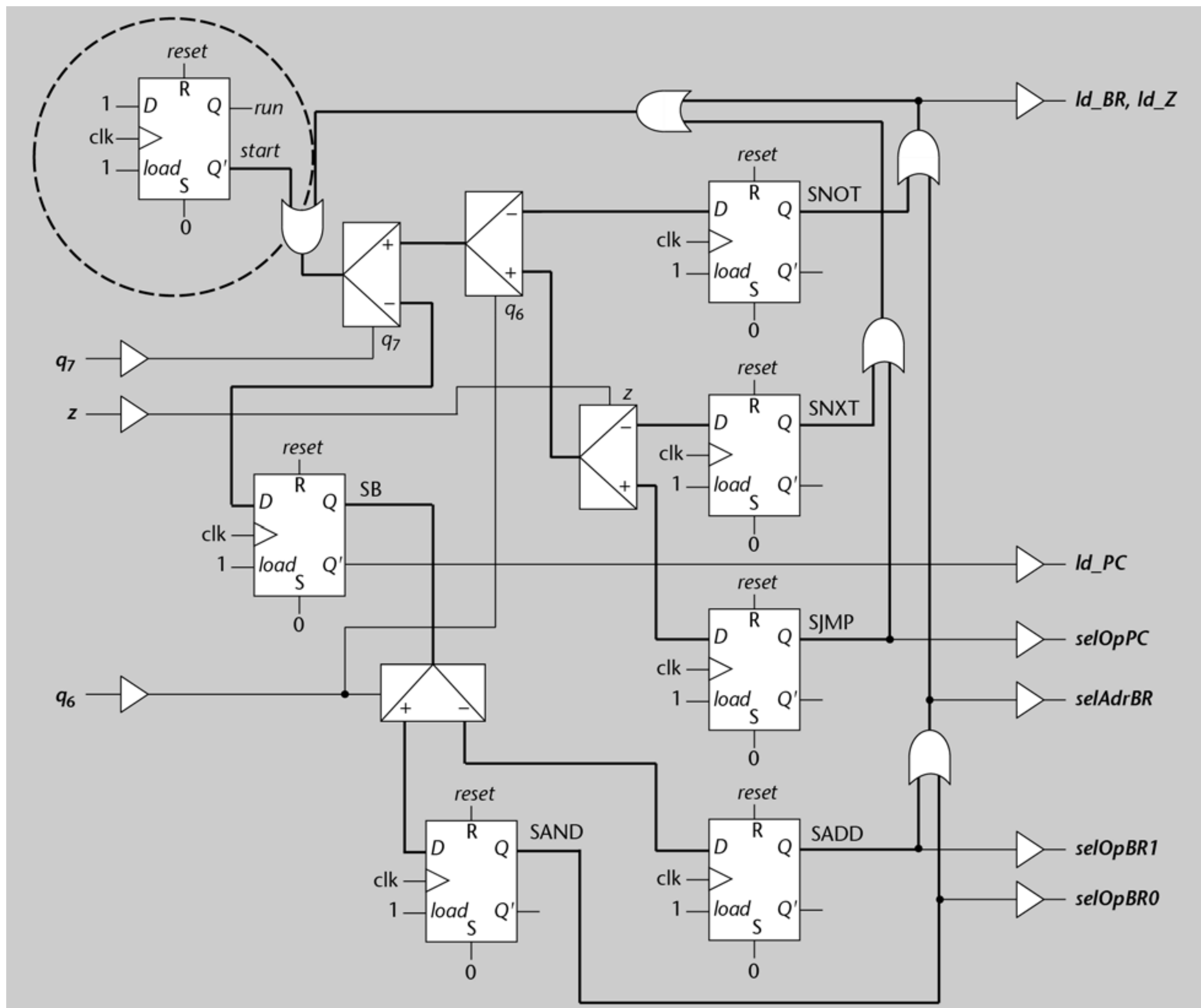
minan con “selOp” previo al nombre del recurso de memoria al que le proporcionan la entrada de datos. En el caso del banco de registros, *selOpBR* hace referencia a la selección del resultado que se debe escribir, *ld_BR*, a la señal que activa la escritura, y *selAdrBR* a la dirección del registro cuyo contenido será presentado en *Dout* (0 para Rf y 1 para Rd). La señal *selOpPC* es 1 para seleccionar la dirección de salto, que es $Q_{5-0}$.

| Estado | *ld_PC* | *selOpPC* | *ld_B* | *selOpBR* | *ld_BR* | *selAdrBR* | *ld_Z* |
|---|---|---|---|---|---|---|---|
| SNXT | 1 | 0 | x | xx | 0 | x | 0 |
| SJMP | 1 | 1 | x | xx | 0 | x | 0 |
| SB | 0 | x | 1 | xx | 0 | 0 | x |
| SNOT | 1 | 0 | x | 00 | 1 | 0 | 1 |
| SAND | 1 | 0 | x | 01 | 1 | 1 | 1 |
| SADD | 1 | 0 | x | 10 | 1 | 1 | 1 |

De cara a la implementación de la unidad de control, la mayoría de las salidas se pueden obtener directamente de un único bit del código de estado, salvo las señales *ld_BR*, *selAdrBR* y *ld_Z*, que se debe construir con sumas lógicas de estos bits.

En la figura 49 aparece el circuito de control del Femtoproc. Es importante tener presente que, a diferencia de las unidades de control vistas en el apartado 2.6, no hay señal de arranque (*start*) ni señal de salida (*stop*): este circuito funciona de manera autónoma ejecutando un bucle infinito (es como se denomina, aunque siempre se puede reiniciar con *reset* o cortar la alimentación de corriente). El circuito que hay rodeado con un círculo de línea discontínua proporciona el primer 1, justo después de un *reset*: el biestable se pone a 0 y, en consecuencia, la salida negada se pone a 1. En el flanco siguiente de reloj, cargará un 1 y *start* pasará a ser 0.

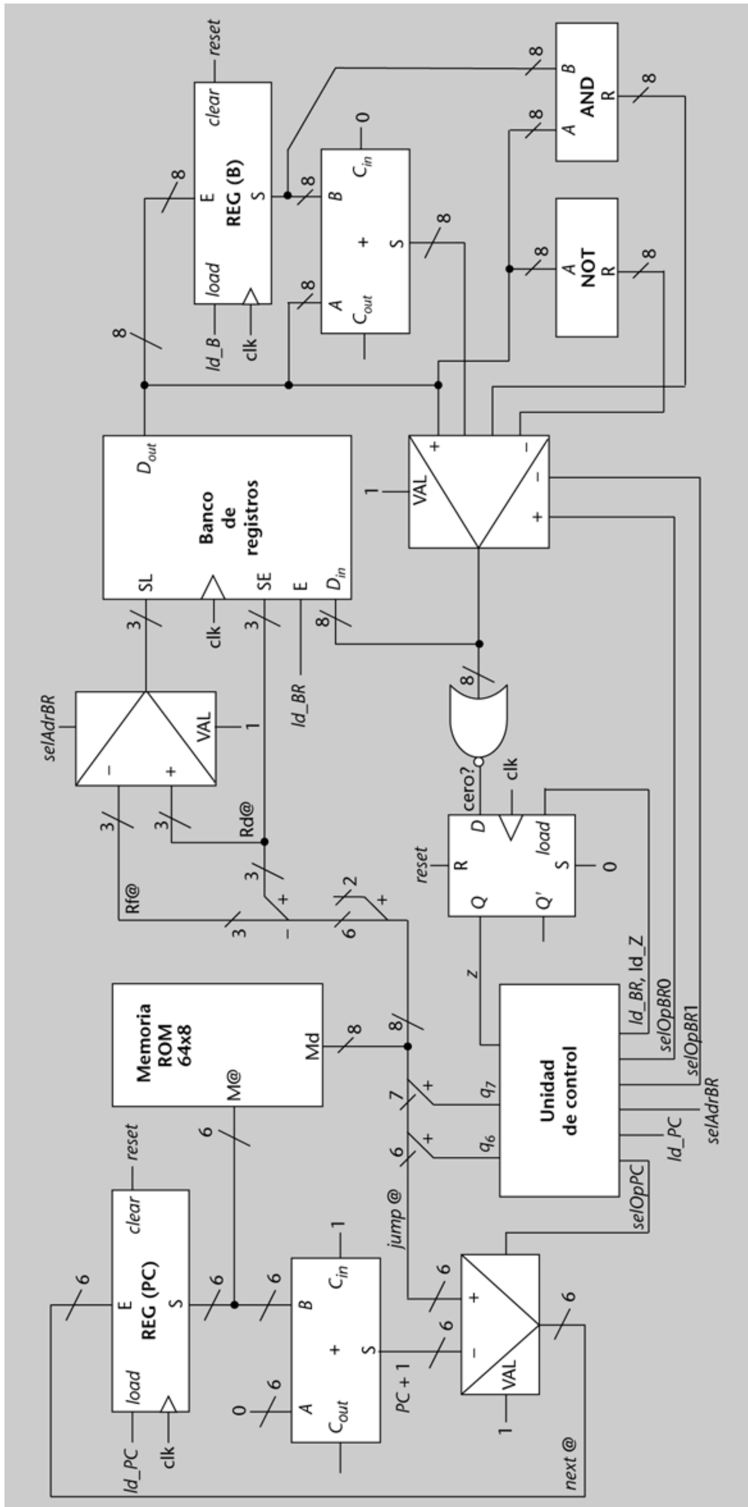
Figura 49. Unidad de control del Femtoproc, con codificación de estados de tipo *one-hot bit*



La unidad de control aprovecha las sumas lógicas que implementan los arcos de retorno en el nodo de decisión de $q_7$ para generar las señales *selAdrBR* e *ld_BR* (o *ld_Z*).

El esquema del circuito completo del Femtoproc se puede ver en la figura 50. Si se observa el dibujo en el sentido de lectura del texto que contiene, a la izquierda de la unidad de control está el circuito para actualizar el *PC*, que depende de *selOpPC*. A la derecha está la parte de operaciones con datos del banco de registros BR: complemento, producto lógico y suma aritmética. En estos dos últimos casos, se toma un operando del registro *B*, que habrá almacenado el contenido de Rf en el ciclo de reloj anterior (con *ld_B* = 1). El multiplexor de buses que hay bajo BR es el que elige qué operación se debe realizar en la instrucción en curso y, por lo tanto, (*selOpBR*1, *selOpBR*0) deciden qué valor se ha de almacenar en BR[Rd] y que se debe comparar con cero para actualizar el biestable correspondiente, que genera la señal *z* para la unidad de control. Por este motivo, *ld_BR* = *ld_Z*.

Figura 50. Esquema completo del Femtoproc, con la unidad de control como módulo



Como se ha comentado, con esta máquina de interpretación se podría ejecutar cualquier programa construido con las instrucciones que puede descodificar. En el ejemplo siguiente se muestra un programa para calcular el máximo común divisor de dos números.

En este caso, se supone que los registros *R*6 y *R*7 se han sustituido por los valores de entrada del programa (es decir, son registros solo de lectura) y que el resultado queda en *R*5. Hay que tener presente que, inicialmente, todos los registros, excepto *R*0, *R*1, *R*6 y *R*7, están a 0.

El programa calcula el máximo común divisor mediante restas sucesivas, según la expresión siguiente:

$$\mathrm{MCD}(a,b) = \begin{array}{l} \mathrm{MCD}(a-b,\, b),\ \text{si } a > b \\ \mathrm{MCD}(a,\, b-a),\ \text{si } b > a \\ a,\ \text{si } b = 0 \\ b,\ \text{si } a = 0 \end{array}$$

La tabla siguiente muestra el contenido de la ROM del Femtoproc para el cálculo del MCD. Las primeras entradas, en las que en la columna "Instrucción" pone "configuración", son de tipo explicativo y no de contenido. Básicamente, establecen qué registros se utilizarán de entrada de datos desde el exterior y cuáles de salida. En el resto de las filas están la dirección y el contenido de la memoria. La dirección se expresa con números en hexadecimal, de aquí que se añada una *h* al final. En el caso de que sea una dirección de salto, también se pone una etiqueta que las identifique para que el flujo de control del programa sea más fácil de seguir. Las instrucciones se representan con los símbolos que se corresponden con las operaciones que hay que realizar (ADD, AND, NOT o JZ) y los que representan los operandos, que pueden ser registros o direcciones. El número de los registros empieza por *R* y lleva un número que identifica la posición en el banco de registros BR.

| Dirección | Instrucción | Codificación | Comentario |
|---|---|---|---|
| --- | `configuración` | – | *R*0 = 1000 0000 = 80h, bit de signo |
| --- | `configuración` | – | *R*1 = 0000 0001 = 01h, bit unitario |
| --- | `configuración` | – | *R*5, registro de salida |
| --- | `configuración` | – | *R*6, registro de entrada |
| --- | `configuración` | – | *R*7, registro de entrada |
| `00h` | `ADD R2, R7` | 00 010 111 | Pasa uno de los valores a un registro de trabajo |
| `01h` | `JZ 12h (end)` | 11 010010 | Si es 0, el MCD es el otro número |
| `02h` | `ADD R3, R6` | 00 011 110 | Pasa el otro valor a un segundo registro auxiliar |
| `03h` | `JZ 12h (end)` | 11 010010 | Si es 0, el MCD es el primer número |
| `sub,04h` | `NOT R4, R2` | 10 100 010 | *R*4 = NOT(*R*2) |
| `05h` | `ADD R4, R1` | 00 100 001 | *R*4 = –*R*2 = Ca2(*R*2) = NOT(*R*2) + 1 |
| `06h` | `ADD R4, R3` | 00 100 011 | *R*4 = *R*3 – *R*2 |
| `07h` | `AND R0, R4` | 01 000 100 | Comprueba el bit de signo, el resultado no se guarda, porque *R*0 es solo de lectura |
| `08h` | `JZ 0Dh (pos)` | 11 001101 | Si es positivo (si *R*3 > *R*2) pasa a 'pos' |
| `09h` | `NOT R2, R4` | 10 010 100 | Si es negativo, se cambia de signo el resultado |
| `0Ah` | `ADD R2, R1` | 00 010 001 | y se deja en *R*2: *R*2 = –(*R*3 – *R*2) |
| `0Bh` | `AND R0, R1` | 01 000 001 | Fuerza a que el resultado sea cero |
| `0Ch` | `JZ 04h (sub)` | 11 000100 | Vuelve a hacer otra resta |
| `pos,0Dh` | `NOT R3, R4` | | |
| `0Eh` | `NOT R3, R3` | | *R*3 = *R*4 = *R*3 – *R*2 |
| `0Fh` | `JZ 12h (end)` | | |
| `10h` | `AND R0, R1` | | |
| `11h` | `JZ 04h (sub)` | | Vuelve a hacer otra resta |

| Dirección | Instrucción | Codificación | Comentario |
|---|---|---|---|
| end,12h | ADD R5, R2 | | |
| 13h | ADD R5, R3 | | *R*5 = *R*2 + *R*3, pero uno de los dos es cero |
| 14h | AND R0, R1 | | Se espera, hasta que se haga 'reset' |
| hlt,15h | JZ 15h (hlt) | | |

Aunque la máquina que se ha visto en este ejemplo sería plenamente operativa, solo serviría para ejecutar programas muy simples: habría que incrementar el repertorio de instrucciones, disponer de más registros para datos y que la memoria de programa fuera mayor.

### Actividades

**12.** Completad la columna de la codificación en la tabla anterior.

**13.** Localizad, en el programa anterior, la secuencia de instrucciones que permite copiar el valor de un registro a otro.

**14.** ¿Con qué secuencia de instrucciones se pueden desplazar los bits de un registro hacia la izquierda?

## 3.2. Máquina elemental

Se denomina **lenguaje máquina** el lenguaje en el que se codifican los programas que interpreta una determinada máquina. El lenguaje máquina de la máquina algorítmica que se ha comentado en el subapartado anterior permite, en teoría, ejecutar cualquier algoritmo que esté descrito en un **lenguaje de alto nivel** de abstracción, como C++ o C, previa traducción. No obstante, excepto para los programas más sencillos, la conversión hacia programas en lenguaje máquina no sería factible por varios motivos:

- El espacio de datos es muy limitado para contener los que se utilizan habitualmente en un programa de alto nivel. Hay que tener en cuenta que, por ejemplo, una palabra se almacena como una serie de caracteres y, según su longitud, podría ocupar fácilmente todo el banco de registros, aunque se ampliara significativamente. De hecho, la palabra "Femtoproc" no se puede guardar en el Femtoproc porque ocupa más de 6 caracteres (o bytes).

- La memoria de programa tiene poca capacidad si se tiene en cuenta que las instrucciones de alto nivel se tienen que llevar a cabo con instrucciones muy simples del repertorio del lenguaje máquina. Por ejemplo, una instrucción de alto nivel de repetición de un bloque de programa se debería convertir en un programa en lenguaje máquina que evaluara la condición de repetición y pasara a la ejecución del bloque. Así, una cosa como:

  ```
  mientras (c < f) hacer B;
  ```

  se debería convertir en un programa que leyera el contenido de las variables *c* y *f*, las comparara y, según el resultado de la comparación, saltara a la primera instrucción del bloque *B* o a la instrucción de alto nivel siguiente.

- El repertorio de instrucciones es demasiado reducido como para poder llevar a cabo operaciones comunes de los programas de alto nivel de modo

eficiente. Por ejemplo, sería muy costoso traducir una suma de una serie de números, ya que se deben tener tantas instrucciones como números que sumar. Un caso más simple es el de la resta. Algo como $R3 = R3 - R2$ necesita tres instrucciones en lenguaje máquina: el complemento de $R2$, el incremento en 1 y, finalmente, la suma de $R3$ con $-R2$.

En una máquina elemental destinada a ejecutar programas hay que resolver bien estos problemas.

El repertorio de instrucciones debe ser más completo, lo que implica que el algoritmo de interpretación sea más complejo y que la propia máquina sea más compleja, ya que debe haber más recursos para poder llevar a cabo todas las operaciones del repertorio. Así, habría que tener, como mínimo, una instrucción de lenguaje máquina para cada una de las operaciones aritméticas, lógicas y de movimiento más habituales. Además, debe incluir una mayor variedad de saltos, incluido uno incondicional. De esta manera, la traducción de programas de más alto nivel de abstracción a programas en lenguaje máquina es más directa y el código ejecutable, más compacto. (Eso sí, hay que tener presente que la máquina interpretadora será más grande y compleja.)

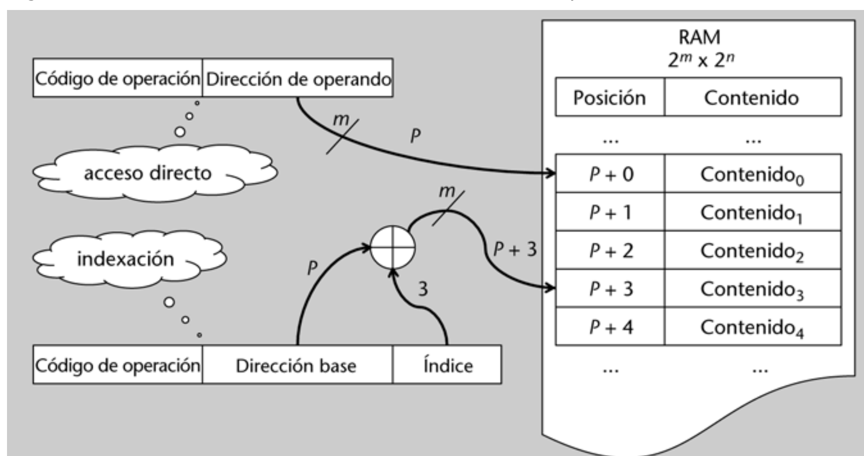
La memoria de programas ha de tener mucha más capacidad y, en consecuencia, las direcciones deben ser de más bits. Hay que tener en cuenta, en este sentido, que la codificación de las instrucciones también se realiza con más bits, ya que el repertorio se ha ampliado.

El espacio reservado a los datos también debe crecer. Por una parte, el número de registros de la máquina se puede ampliar, pero resulta complejo decidir hasta qué punto. La solución, además de esta ampliación, consiste en utilizar también una memoria para los datos.

**Acceso a datos en memoria**

Para el acceso a los datos en memoria se puede utilizar un acceso directo con su dirección. Conviene que haya, además, mecanismos de acceso indexado, en los que sea posible acceder a una serie de datos mediante una dirección base y un índice para hacer referencia a un dato concreto de la serie. De esta manera, la manipulación de los datos por parte de los programas en lenguaje máquina es más eficiente. En la figura 51 hay un ejemplo de cada uno, con un código de instrucción que incluye el código de la operación que hay que realizar, la dirección del operando (*P*) y, para el acceso indexado, un índice que vale, en el ejemplo, 3.

Figura 51. Ilustración de los accesos directo e indexado al operando de una instrucción



Si tomamos como referencia el caso del Femtoproc, la memoria de programación es de tipo ROM. Las memorias para datos, en cambio, deben permitir tanto lecturas como escrituras. En este sentido, si se tuviera que mejorar esta máquina, se le deberían incorporar memorias para instrucciones y para datos diferenciados.

Esta manera de construir las máquinas, con una memoria para las instrucciones y otra para los datos, se denomina **arquitectura Hardvard**.

**Arquitectura Hardvard**

La arquitectura Hardvard se denomina así porque refleja el modelo de construcción de una de las primeras máquinas computadoras que se construyó: la IBM Automatic Sequence Controlled Calculator (ASCC), denominada *Mark I*, en la Universidad de Hardvard, alrededor de 1944. Esta máquina tenía físicamente separado el almacenaje de las instrucciones (en una cinta perforada) del de los datos (en una especie de contadores electromecánicos).

En ocasiones, sin embargo, puede resultar más sencillo tener tanto las instrucciones como los datos en la misma memoria. De esta manera, la máquina solo ha de gestionar los accesos a una única memoria y, adicionalmente, puede ejecutar tanto programas pequeños que utilicen muchos datos como otros muy grandes que traten con pocos.

Las máquinas construidas de manera que utilizan una misma memoria para datos e instrucciones siguen la **arquitectura de Von Neumann**.

**Arquitectura de Von Neumann**

La arquitectura de Von Neumann se describe en *First Draft of a Report on the EDVAC*, fechado a 30 de junio de 1945. Este informe recogía la idea de construir computadores con un programa almacenado en memoria en lugar de hacerlo como máquinas para algoritmos específicos. De hecho, describe el estado del arte en la época y propone la construcción de una máquina denominada *Electronic Discrete Variable Automatic Computer* ("computador electrónico automático de variables discretas"), que se basa, sobre todo, en el trabajo hecho en torno a otro computador primerizo: el ENIAC o *Electronic Numerical Integrator And Computer* ("integrador numérico y calculador electrónico").

Con independencia de la arquitectura que se siga para materializar estas máquinas, es necesario que puedan interpretar un repertorio de instrucciones suficiente para permitir una traducción eficiente de los programas en lenguajes de alto nivel y que tengan una capacidad de memoria lo suficientemente grande para introducir tanto las instrucciones como los datos de los programas que deben ejecutar.

### 3.2.1. Máquinas algorítmicas de unidades de control

Una máquina de este estilo todavía se puede ver como una máquina algorítmica de interpretación, pero el algoritmo que interpreta las instrucciones del lenguaje máquina es bastante más complejo que el que se ha visto anteriormente. En líneas generales, el intérprete trabaja de manera similar, pero las fases del ciclo de ejecución de las instrucciones se alargan hasta tardar varios periodos de reloj. Es decir, el número de pasos para llegar a ejecutar una única instrucción de un programa en lenguaje máquina es elevado.

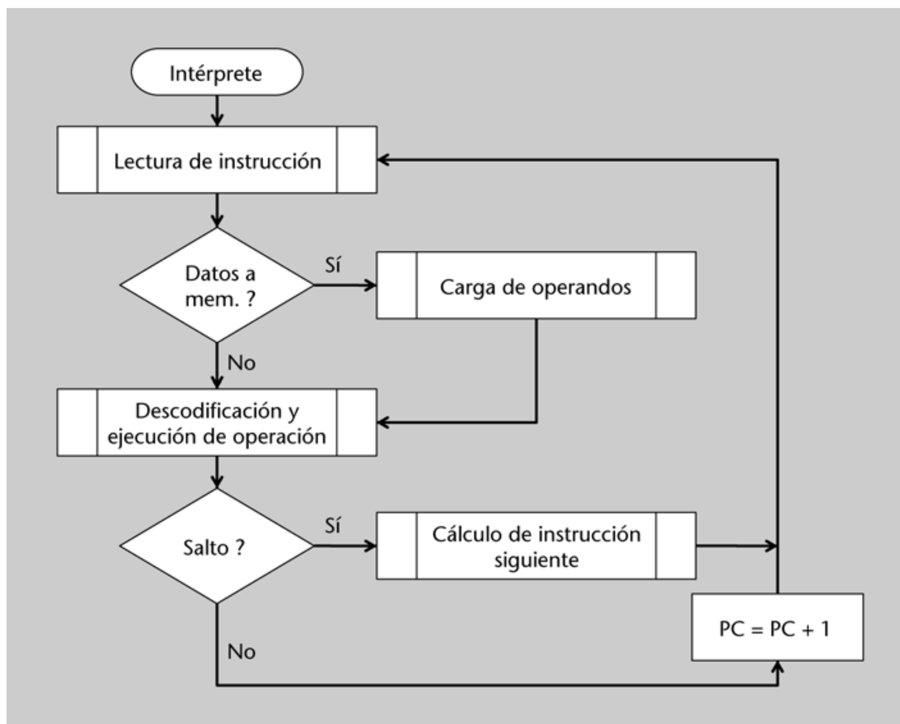
Hay que tener presente que la lectura de una sola instrucción puede necesitar varios pasos, dado que la anchura en bits debe permitir codificar un repertorio grande, lo que provoca que pueda ocupar más de una palabra de memoria.

Además, en el caso de que se trate de instrucciones que trabajen con datos en memoria, hay que hacer su lectura, que puede no ser directa y, por lo tanto, prolongar una serie de pasos que se incluyen en la fase de carga de operandos.

Como también es conveniente que el abanico de operaciones que puedan llevar a cabo los computadores sea bastante grande, la descodificación de éstas para poder ejecutarlas y su propia ejecución suelen requerir una serie de pasos que depende del tipo de operación y, en consecuencia, existe una diferencia de tiempo entre la ejecución de unas y otras instrucciones. La ampliación de la variedad de saltos también complica de manera similar el cálculo de la instrucción siguiente.

La figura 52 resume el flujo de control de una unidad de procesamiento que implementa uno de estos repertorios de instrucciones. Cada etapa o fase se debe resolver con una serie de pasos que se alargan durante unos cuantos ciclos de reloj. Hay que tener en cuenta que se trata de nodos de proceso predefinido y, por lo tanto, que se corresponden con esquemas de cálculo o, incluso, máquinas algorítmicas completas.

Figura 52. Diagrama de flujo del ciclo de ejecución de instrucciones




La implementación de estas máquinas algorítmicas de interpretación se podría hacer de modo similar a como se ha visto, a partir de los diagramas de flujo correspondientes, pero teniendo en cuenta que la secuenciación de los pasos que se deben seguir es muy compleja. Por lo tanto, la unidad de control tiene un número de estados enorme que hace poco práctica la implementación como circuito combinacional basado en bloques lógicos y registros.

En la máquina algorítmica anterior, correspondiente al caso del Femtoproc, la ejecución de las instrucciones dura entre uno y dos ciclos de reloj. Por lo tanto, las secuencias de pasos son bastante simples. En máquinas que interpreten instrucciones más complejas, las secuencias se alargan. Si esto se une al hecho de que la cantidad de combinaciones de entradas posibles para la parte con-

troladora crece exponencialmente, ya que los repertorios de instrucciones son mayores (más bits para codificar las operaciones correspondientes) y se utilizan muchos más bits de condición de la parte operativa (indicación de cero, acarreo o *carry*, desbordamiento, etc.), el número potencial de estados crece de la misma manera.

Por este motivo, es conveniente modelar la parte de control de estas máquinas algorítmicas con otras máquinas algorítmicas encargadas de interpretar las primeras. En este caso, también serán máquinas algorítmicas de interpretación de instrucciones, pero de un repertorio más limitado. Por ejemplo, una instrucción para cada tipo de nodo en un diagrama de flujo.

Para distinguir esta máquina de la máquina que interpreta el repertorio de instrucciones del procesador, se habla de la máquina que interpreta las microinstrucciones de la unidad de control.

Las **microinstrucciones** son aquellas operaciones que se realizan con los recursos de cálculo de una unidad de procesamiento en un ciclo de reloj. Cada una de las órdenes que se da a los elementos de la unidad de procesamiento se denomina **microorden**. El programa de interpretación de las instrucciones del lenguaje máquina se conoce, obviamente, como **microprograma**.

Los procesadores que interpretan repertorios de instrucciones complejos suelen estar microprogramados, es decir, la unidad de control correspondiente es una máquina algorítmica implementada con un intérprete de microprogramas y el microprogama correspondiente.

### 3.2.2. Máquinas algorítmicas microprogramadas

Como se ha comentado, el repertorio de microinstrucciones suele ser muy reducido. Por ejemplo, puede tener dos tipos, con correspondencia con los nodos de procesamiento y decisión de las máquinas algorítmicas:

**1)** Las de **procesamiento** son las que llevan a cabo alguna operación con el camino de datos, es decir, las que efectúan un conjunto de microórdenes en un único ciclo de reloj. Una microinstrucción común de este primer tipo es la que reúne las microórdenes necesarias para cargar en un registro de destino el resultado de una operación realizada con datos provenientes de varios registros fuente.

**2)** Las de **decisión** sirven para hacer saltos condicionales. Por ejemplo, una microinstrucción de salto condicional activaría las microórdenes para seleccionar qué condición se debe cumplir y la de carga del contador de programa.

También existe la opción de tener un repertorio basado en un tipo único de microinstrucción que consista en un salto condicional y un argumento variable que indique qué microórdenes se deben ejecutar en el ciclo correspondiente.

En la tabla siguiente encontramos un ejemplo de un posible formato de un repertorio como el del primer caso. Hay que tener presente que el número de bits debe ser tan grande como para que quepan todas las posibles microórdenes del camino de datos y las direcciones de la memoria de microprogramación, que ha de tener capacidad suficiente para almacenar el microprograma de interpretación del repertorio de instrucciones.

<table>
<tr><th rowspan="2">Microinstrucción</th><th colspan="16">Bits (1 por microorden)</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>EXEC</td><td>0</td><td colspan="15">selectores de entrada, cargas de registros, operación de la ALU...</td></tr>
<tr><td>JMPIF</td><td>1</td><td colspan="4">selectores de condición</td><td>–</td><td>–</td><td colspan="9">dirección de salto (microinstrucción siguiente)</td></tr>
</table>

El camino de datos se supone organizado de una manera similar a la de un esquema de cálculo con varios registros y un único recurso de cálculo programable, para el que hay que proporcionar la operación que se realiza en una microinstrucción concreta.

La arquitectura del camino de datos que se ha visto y que se basa en el uso de un único recurso de cálculo programable es bastante común en la mayoría de los procesadores. Como este recurso hace tanto operaciones aritméticas (suma, resta, cambio de signo, incrementos, etc.) como lógicas (suma y producto lógico, complemento, desplazamientos de bits, etc.), se hace referencia a él como la unidad aritmeticológica (o ALU, del inglés, *arithmetic logic unit*) del camino de datos del procesador.

La máquina algorítmica que interprete un repertorio de microinstrucciones como el anterior permitiría ejecutar un microprograma de control de otra máquina algorítmica, que sería, finalmente, el procesador del repertorio de instrucciones de un lenguaje máquina concreto. Dado que se trata de una máquina algorítmica muy sencilla que se ocupa de ejecutar las microinstrucciones en una secuencia determinada, se suele denominar **secuenciador**.

Aunque el secuenciador con EXEC y JMPIF es funcionalmente correcto, hay que tener presente que, durante la ejecución del JMPIF no se lleva a cabo ninguna operación en la unidad de procesamiento. Hay varias opciones para aprovechar este ciclo de reloj por parte del camino de datos. Una de las más sencillas es que el secuenciador trabaje con un reloj el doble de rápido que el de la unidad de procesamiento que controla e introducir un ciclo de espera entre EXEC y EXEC. Otra consiste en aprovechar los bits que no se aprovechan (por ejemplo, en la tabla anterior, el JMPIF no utiliza los bits en posición 9 y 10) como bits de codificación de determinadas microórdenes. Así, con el JMPIF también se llevarían a cabo algunas operaciones en la unidad de procesamiento. (De hecho, es un caso como el del repertorio de tipo único, en el que todas las microinstrucciones incluyen microórdenes y direcciones de salto). Por lo tanto, el problema de no aprovechar la unidad de procesamiento durante los saltos se minimiza.

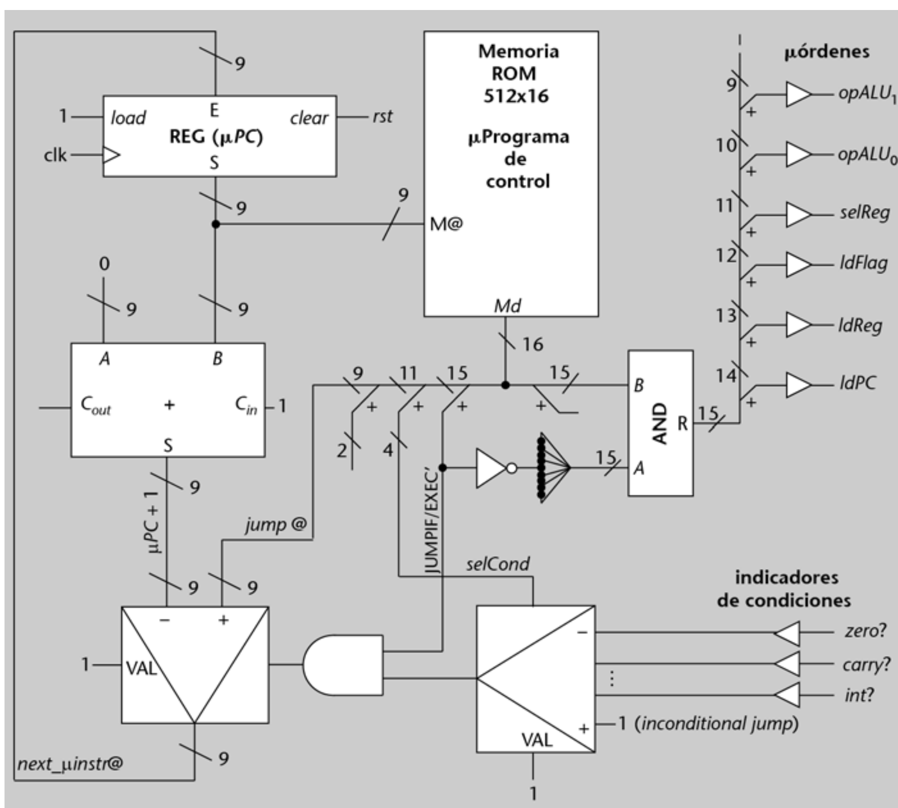
En la figura 53 aparece el esquema de un circuito secuenciador para el repertorio de microinstrucciones que se acaba de comentar. Se trata de un secuenciador paralelo, en el que cada microorden tiene un bit asociado. Por este

motivo, este tipo de secuenciadores emplea memorias con anchuras de palabra muy grandes. En la práctica, suele compensar tener una codificación más compacta que reduzca la anchura a cambio de utilizar circuitos para descodificar las microinstrucciones.

La memoria ROM contiene una codificación binaria del microprograma que ejecuta una máquina algorítmica de interpretación de instrucciones particular. Cada palabra de la memoria sigue el formato que se ha detallado en la tabla anterior.

En el caso de que se trate de una microinstrucción de tipo EXEC, los bits $Md_{14-0}$ son las microórdenes que están asociadas. Las señales correspondientes se han identificado con nombres que ejemplifican los tipos de microórde-nes que puede haber en una máquina-tipo: *ldPC* para cargar un nuevo valor en el contador de programa; *ldReg* para que un registro guarde, al acabar el ciclo de reloj actual, algún nuevo resultado, que se elige con *selReg*; $opALU_0$ y $opALU_1$ para seleccionar la operación que debe realizar la ALU; *ldFlag* para actualizar los biestables que almacenan condiciones sobre el último resultado (si ha sido cero, si se ha producido acarreo, etc.), y así hasta 15. Al mismo tiempo que se suministran estas señales a la parte operacional, se calcula la dirección de la siguiente microinstrucción, que es la que se encuentra en la posición de memoria siguiente. Para ello, el contador de microprograma (*μPC*) se incrementa en 1: el multiplexor que genera *next_μinst@* selecciona la entrada *μPC* + 1 cuando se está ejecutando una microinstrucción EXEC.

Figura 53. Esquema de un circuito secuenciador de microinstrucciones



Cuando se ejecuta un JMPIF todas las microórdenes están inactivas porque se hace un producto lógico con el bit que indica si es JMPIF o EXEC, de manera que, cuando JMPIF/EXEC' ($Md_{15}$) sea 1, las microórdenes serán 0. En este caso, los bits $Md_{14-0}$ no son microórdenes, sino que representan, por una parte, la selección de la condición (*selCond,* que es $Md_{14-11}$) que hay que tener en cuenta para efectuar o no el salto en la secuencia de microinstrucciones y, por otra, la dirección de la microinstrucción siguiente (*jump@*, que es $Md_{8-0}$) en el caso de salto. La señal *selCond* se ocupa de elegir qué bit de condición proveniente del camino de datos se debe tener en cuenta a la hora de dar el salto. Como se puede ver en la figura, estos bits pueden indicar si el último resultado ha sido cero (*zero*?), si se ha producido acarreo (*carry*?) o si hay que interrumpir la ejecución del programa (*int*?), por ejemplo. Hay que fijarse en que una de las entradas del multiplexor correspondiente es 1. En este caso, la condición siempre se cumplirá. Esto sirve para dar saltos incondicionales. La decisión de efectuar o no el salto depende de la condición seleccionada con *selCond* y de que se esté ejecutando una microinstrucción JMPIF, de aquí que exista la puerta AND previa a la generación del control del multiplexor que genera *next_µinst@*.

Con un secuenciador como este sería posible construir controladores para máquinas algorítmicas relativamente complejas: sus diagramas de flujo podrían llegar a tener hasta 512 nodos de condición o procesamiento.

Para hacer una máquina capaz de ejecutar programas de propósito general que siga la arquitectura de Von Neumann, es habitual implementar las máquinas algorítmicas de interpretación del repertorio de instrucciones correspondiente con una unidad de control microprogramada.

### 3.2.3. Una máquina con arquitectura de Von Neumann

A modo de ejemplo, se presenta un procesador sencillo de arquitectura Von Neumann que utiliza pocos recursos en el camino de datos y una unidad de control microprogramada. Es, de hecho, otro de los muchos procesadores sencillos que hay por el mundo y, por ello, se denomina **YASP** (del inglés, *yet another simple processor*).

YASP trabaja con datos e instrucciones de 8 bits, que deben estar almacenadas en la memoria principal o provenir del exterior por medio de algún puerto de entrada (un registro que carga información que proviene de algún periférico de entrada).

La memoria principal es de solo 256 bytes, por lo que basta con un único byte para representar todas las posiciones de memoria posibles, desde la 0 hasta la 255.

El repertorio de instrucciones que entiende YASP consta de las que hacen operaciones aritmeticológicas, las de movimiento de información, las de manipu-

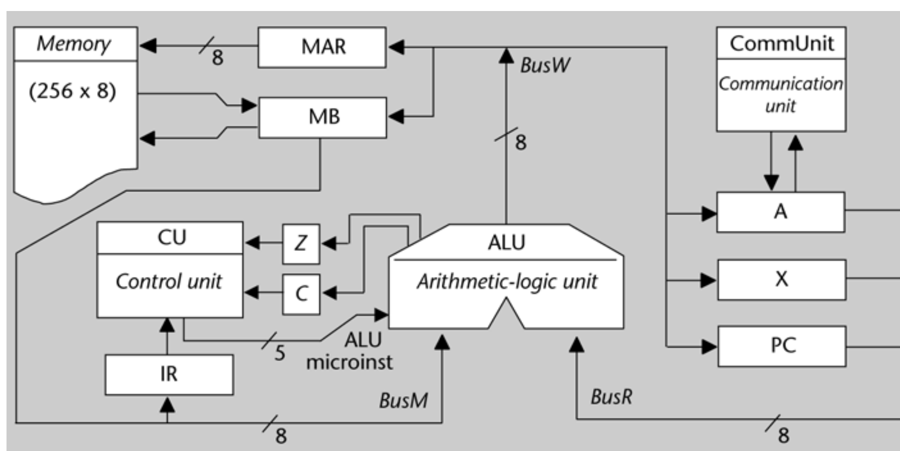
lación de bits y, finalmente, las de control de flujo. Como es habitual, las operaciones aritméticas se efectúan considerando que los números se representan en complemento a 2.

La codificación de las instrucciones incluye campos (grupos de bits) para identificar el tipo de instrucción, la operación que debe realizar la ALU y los operandos que participan, entre otros. El argumento para acceder al segundo operando, si es necesario, se encuentra en un byte adicional. Así, hay instrucciones que ocupan un byte y otras, dos.

Los formatos de las instrucciones de los lenguajes máquina suelen ser muy variables porque ajustan las amplitudes de bits al tipo de operación que realizan y a los operandos que necesitan. Así, es habitual tener codificaciones en las que haya instrucciones que ocupen un único byte con otras que necesiten dos, tres, cuatro o más. En el lenguaje máquina del YASP, algunas ocupan un byte, si no necesitan ningún dato adicional, y otras ocupan dos. En el primer caso se encuentran, por ejemplo, las instrucciones que trabajan con registros. En el segundo, las que trabajan con datos en memoria.

En el esquema de YASP que aparece a continuación se pueden ver todos los elementos que forman parte de él y el modo como están conectados. Hay tres buses de conexión: el de memoria (*BusM*), el de registros (*BusR*) y, finalmente, el de resultados (*BusW*), que pueden ser transportados hacia la memoria o hacia los registros. La comunicación con el exterior se lleva a cabo mediante el registro *A* y un módulo específico (CommUnit), que también está gobernado por la unidad de control (CU). No se muestran las señales de control. Las señales de entrada de la CU son las del código de la instrucción en curso, que se almacena en IR, y los indicadores de cero (Z) y de acarreo (C).

Figura 54. Diagrama de bloques del procesador elemental YASP




Para poder procesar los datos, dispone de un registro de 8 bits denominado **acumulador** (o registro A), ya que es el registro en el que se acumularía el resultado de la suma de una secuencia de bytes. De hecho, el resultado de cualquier operación aritmética o lógica se guarda en este registro.

Estas operaciones se llevan a cabo en la ALU, que toma un posible segundo operando de la memoria principal mediante el registro MB, de *memory buffer*. Hay dos elementos de memoria adicionales de un bit cada uno para indicar si

el resultado de la operación ha sido cero o no (el bit Z), y si la operación ha generado acarreo o no (el bit C).

Para hacer las operaciones de acceso a este segundo operando y, en general, para acceder a cualquier posición de memoria, hay que guardar la dirección en el registro de direcciones de memoria (MAR, del inglés *memory address register*) antes de efectuar la lectura del contenido de memoria o de escribir el contenido de MB.

Como la dirección del segundo operando no siempre se conoce de antemano, también se puede dar la dirección de una posición de memoria en la que estará guardada la dirección de este operando. Es lo que se conoce como **indirección**, por el hecho de no dar directamente la dirección del operando.

A veces resulta conveniente trabajar con una serie de datos de los que se conoce la dirección inicial; es decir, la localización del primer dato de la serie. En este caso, hay que sumar a la dirección inicial la posición del dato con el que se quiere trabajar. Para hacer esta indexación, YASP dispone de un registro auxiliar adicional, el registro X. Así, la dirección del operando se calcula sumando a la dirección base (la dirección del primer dato) el contenido del registro X.

En resumen, las instrucciones del lenguaje máquina de YASP tienen cuatro modos diferentes de obtener la dirección del segundo operando de una operación diádica (con dos operandos, uno de los cuales es el registro A):

**1) Inmediata**. El valor del operando se especifica junto con la operación.

**2) Directa**. La dirección del operando se indica con la operación. Hay que buscar el valor en la posición indicada.

**3) Indirecta**. La posición de memoria en la que está el operando está guardada en la dirección que acompaña la operación. Es decir, se da la dirección en la que está la dirección real del operando.

**4) Indexada**. La dirección del operando se obtiene sumando el registro índice X a la dirección que se adjunta con la operación.

Más formalmente, las direcciones de los operandos se conocen como **direcciones efectivas** para distinguirlas de las que se dan en las instrucciones del programa que se ejecuta; mientras que las formas de obtener las direcciones efectivas se conocen como **modos de dirección**. En concreto, se han descrito los modos inmediato, directo, indirecto e indexado.

Una vez obtenido el segundo operando, la ALU puede realizar la operación entre este y el contenido del registro A. El resultado se deja normalmente en el mismo registro. Si la operación solo necesita un operando, entonces A es el único operando y el resultado de la operación monádica se guarda a continuación.

En el caso de esta máquina elemental, la ALU realiza sumas, restas, cálculos del opuesto (cambios de signo), incrementos, decrementos, *y*s lógicas, *o*s inclusivas y exclusivas, desplazamientos y rotaciones. También permite el paso directo de la información, sin realizar ninguna operación.

Normalmente, la instrucción siguiente se encuentra a continuación de la que se acaba de ejecutar; es decir, basta con incrementar el contenido del PC en uno o en dos, según la instrucción que se ejecute. A veces, sin embargo, interesa ir a un punto diferente del programa. Este salto puede ser incondicional, o condicional, si depende del estado de la máquina, que, en este caso, viene definido por los bits C y Z.

En el caso de los saltos condicionales, solo se reserva un byte para su codificación y solo hay 5 bits para expresar la dirección de salto, ya que los otros 3 son necesarios para codificar la operación. Por lo tanto, no se puede saltar a cualquier posición de memoria, sino que solo se puede ir a una instrucción situada 16 posiciones adelante o 15 hacia atrás. Esto es así porque se suma un valor en Ca2 en el rango [–16, 15] en el PC incrementado. Estas instrucciones utilizan un **modo de dirección relativo**, es decir, la dirección que se adjunta se suma al contenido del PC. Por lo tanto, la dirección de la instrucción a la que se salta se da en relación con la dirección de la instrucción siguiente a la del salto condicional.

Sea como sea, la ejecución de las instrucciones del lenguaje máquina de YASP se alargan unos cuantos ciclos de reloj, en los que se ejecuta el microprograma correspondiente. Por este motivo, el código de operación de la instrucción en curso se guarda en el registro de instrucciones (IR), de manera que la unidad de control pueda decidir qué debe hacer y en qué orden.

Aunque YASP es un procesador sencillo, es capaz de interpretar un repertorio suficientemente completo de instrucciones gracias a una unidad de control bastante compleja, que se puede materializar de manera directa si se microprograma.

Utilizando un lenguaje de transferencia de registros (RTL, del inglés *register transfer language*), se muestra a continuación el microprograma correspondiente a la ejecución de una instrucción de suma entre A y un operando en memoria.

En la columna "Etiqueta" aparece el símbolo que identifica una posición concreta de la memoria de microprogramas.

La columna que contiene las microinstrucciones detalla qué transferencias de registros se realizarán en cada ciclo de reloj si se trata de EXEC. Las cargas simultáneas de valores en MB y en PC son posibles porque utilizan buses diferentes. De hecho, MB se carga con el valor de la entrada de datos que proviene de la memoria.

En el caso de las microinstrucciones de tipo JMPIF, el primer argumento es la condición que se comprueba, y el segundo, la dirección de salto en el caso de que sea cierto.

En este ejemplo aparece la secuencia completa de microinstrucciones para ejecutar una operación de suma del lenguaje máquina de YASP.

| Etiqueta | Código de operación | μ–instrucción | Comentario |
|---|---|---|---|
| START: | EXEC | `MAR = PC` | **Fase 1**. Lectura de la instrucción |
| | EXEC | `MB = M[MAR], PC = PC + 1` | M[MAR] hace referencia al contenido de la memoria en la posición MAR |
| | EXEC | `IR = MB` | |
| | JMPIF | `2nd byte?, DYADIC` | Descodifica si hay que leer un segundo byte |
| | | ... | |
| DYADIC: | EXEC | `MAR = PC` | **Fase 2**. Cálculo del operando |
| | EXEC | `MB = M[MAR], PC= PC +1` | |
| | JMPIF | `Immediate?, DO` | Direccionamiento inmediato, operando leído |
| | JMPIF | `Not indexed?, SKIP` | |
| | EXEC | `MB = MB + X` | Direccionamiento indexado |
| SKIP: | EXEC | `MAR = MB` | |
| | EXEC | `MB = M[MAR]` | Direccionamiento directo o indexado |
| | JMPIF | `Not indirect?, DO` | |
| | EXEC | `MAR = MB` | |
| | EXEC | `MB = M[MAR]` | Direccionamiento indirecto |
| DO: | JMPIF | `ADD?, X_ADD` | **Fase 3**. Descodificación de operación y ejecución |
| | | ... | |
| X_ADD: | EXEC | `A = A + MB` | |
| | JMPIF | `Inconditional, START` | Bucle infinito de interpretación |
| | | ... | |

La gran ventaja de las unidades de control microprogramadas es la facilidad de desarrollo y mantenimiento posterior. En el ejemplo no se han puesto los códigos binarios correspondientes, pero la tabla del microprograma es suficiente para obtener el contenido de la memoria de microprogramación del secuenciador asociado.

### Actividades

**15.** Sin considerar las operaciones posibles de la ALU, que se codifican con 5 bits, tal y como se muestra en la figura 54, indicad qué microórdenes serían necesarias para controlar la unidad de procesamiento del YASP. Por ejemplo, habría que tener una para que el registro PC cargue el dato del *BusW*, que se podría denominar *ld_PC*.

**16.** Con el esquema del YASP y tomando el microprograma de la suma como ejemplo, haced uno para una instrucción que incremente el contenido del registro X. Podéis suponer que la ALU puede hacer algo como *BusW* = *BusR* + 1.

### 3.3. Procesadores

Como ya se ha comentado en la sección 3.1.1, las máquinas algorítmicas dedicadas a interpretar programas son, de hecho, máquinas que procesan los datos de acuerdo con las instrucciones de sus programas, es decir, son **procesadores**.

La implementación de los procesadores es muy variada, ya que depende del repertorio de instrucciones que deban interpretar y de la función de coste que se quiera minimizar.

En esta sección se explicarán las diferentes organizaciones internas de los procesadores y se hará una introducción a las que se utilizan en los de propósito general y en los que están destinados a tareas más específicas.

#### 3.3.1. Microarquitecturas

Las **microarquitecturas** (en ocasiones se abrevian como *μarch or uarch*, en inglés) son los modelos de construcción de un determinado repertorio de instrucciones en un procesador o, si se quiere, de una determinada arquitectura de un conjunto de instrucciones (*instruction set architecture* o ISA, en inglés). Hay que tener presente que una ISA se puede implementar con diferentes microarquitecturas y que estas implementaciones pueden variar según los objetivos que se persigan en un determinado diseño o por cambios tecnológicos. (La arquitectura de un computador es la combinación de la microarquitectura y de la ISA.)

En general, todos los programas presentan unas ciertas características de "localidad", es decir, de trabajar localmente. Por ejemplo, en un trozo de código concreto, se suele trabajar con un pequeño conjunto de datos con iteraciones que suelen hacerse dentro del propio bloque de instrucciones. En este sentido, es razonable que las instrucciones trabajen con datos almacenados en un banco de registros y que las haya que faciliten la implementación de varios modos de iteración. En todo caso, un determinado repertorio de instrucciones irá bien para un determinado tipo de programas, pero puede no ser tan eficiente para otros.

Hay arquitecturas de instrucciones que requieren muchos recursos porque cubren un rango de operaciones muy amplio (por ejemplo, con aritmética entera y de coma flotante, con multitud de operadores relacionales, con operaciones aritméticas adaptadas a la programación, como incrementos y decrementos, etc.) y porque utilizan muchas variables (por ejemplo, para poder almacenar datos temporales o para servir de ayuda a estructuras de control). En general, las máquinas que implementan estas ISA se conocen como CISC, del inglés *complex instruction set computer* (computador con conjunto de instrucciones complejo).

En contraposición, hay repertorios de instrucciones que requieren menos recursos y son más fáciles de descodificar. Normalmente, son bastante reducidos en comparación con los CISC, por lo que se los denomina RISC, del inglés *reduced instruction set computer* (computador con conjunto de instrucciones reducido).

En general, los RISC tienen una arquitectura basada en un único tipo de instrucción de lectura y almacenamiento. En otras palabras, tienen una única instrucción para lectura y otra para escritura de/en memoria, aunque tengan variantes según los operandos que admitan. A este tipo de arquitecturas se las denomina ***load/store***, y suelen oponerse a otras en las que las instrucciones combinan las operaciones con los accesos a memoria.

Se podría decir, de alguna manera, que el Femtoproc, que se ha visto en el apartado 3.1.1, es una máquina RISC y que el YASP, que se ha explicado en el apartado 3.2.3, es una máquina CISC.

Tanto los RISC como los CISC se pueden construir con microarquitecturas similares, ya que tienen problemas comunes: acceso a memoria por lectura de instrucciones, acceso a memoria por lectura/escritura de datos y procesamiento de los datos, entre otros.

### 3.3.2. Microarquitecturas con *pipelines*

Para ir más rápido en el procesamiento de las instrucciones, se pueden tratar varias formando un ***pipeline***: una especie de cañería en la que pasa un flujo que es tratado en cada uno de sus segmentos, de modo que no hay que esperar a que se procese totalmente una porción del flujo para tratar otra diferente.

En las unidades de procesamiento, los *pipelines* suelen formarse con las diferentes fases del ciclo de ejecución de una instrucción. En la figura 55, hay un *pipeline* de una máquina similar a YASP, pero limitando los operandos a los inmediatos y directos.

Así, el ciclo de ejecución de una instrucción sería:

**1) Fase 1.** Lectura de instrucción (LI).

**2) Fase 2.** Lectura de argumento (LA), que puede estar vacía, si no hay argumento.

**3) Fase 3.** Cálculo de operando (CO), que hace una nueva lectura si es directo.

**4) Fase 4.** Ejecución de la operación (EO), que puede realizar varias cosas, según el tipo de instrucción: un salto, actualizar el acumulador o escribir en memoria, por ejemplo.

Con esta simplificación, cada fase se correspondería con un ciclo de reloj y, por lo tanto, una unidad de procesamiento sin el *pipeline* tardaría 4 ciclos de reloj en ejecutar una instrucción. Como se puede observar, con el *pipeline* se podría tener una acabada en cada ciclo de reloj, excepto la latencia inicial de 4 ciclos.

Figura 55. *Pipeline* de 4 etapas

| Instrucción | Etapa del *pipeline* (fase del ciclo) | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | LI | LA | CO | EO | | | |
| 2 | | LI | LA | CO | EO | | |
| 3 | | | LI | LA | CO | EO | |
| 4 | | | | LI | LA | CO | EO |
| 5 | | | | | LI | LA | CO |
| Período | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | Latencia | | | | | | |

Por lo tanto, los *pipelines* son una opción para aumentar el rendimiento de las unidades de procesamiento y, por este motivo, la mayoría de las microarquitecturas los incluyen.

Ahora bien, también presentan un par de inconvenientes. El primero es que necesitan registros entre etapa y etapa para almacenar los datos intermedios. El último es que tienen problemas a la hora ejecutar determinadas secuencias de instrucciones. Hay que tener en cuenta que no es posible que una de las instrucciones del *pipe* modifique otra que también está en el mismo *pipe*, y que es necesario "vaciar" el *pipe* al terminar de ejecutar una instrucción de salto que lo haga. En este último caso, la consecuencia es que no siempre se obtiene el rendimiento de una instrucción acabada por ciclo de reloj, ya que depende de los saltos que se hagan en un programa.

**Actividad**

**17.** Rehaced la tabla de la figura 55 suponiendo que la segunda instrucción es un salto condicional que sí se efectúa. Es decir, que después de la ejecución de la instrucción la siguiente es otra diferente de la que se encuentra a continuación. De cara a la resolución, suponed que la secuencia de instrucciones es (1, 2, 11, 12...), es decir, que salta de la segunda a la undécima instrucción.

### 3.3.3. Microarquitecturas paralelas

El aumento del rendimiento también se puede conseguir incrementando el número de recursos de cálculo de la unidad de procesamiento para poder realizar varias operaciones de manera simultánea. Por ejemplo, se pueden incluir varias ALU que faciliten la ejecución paralela de las distintas etapas de un *pipeline* o de varias operaciones de una misma instrucción. En este último caso, el paralelismo puede ser implícito (depende de la instrucción) o explícito (la instrucción incluye argumentos que indican qué operaciones y con qué datos

se hacen). Para el paralelismo explícito, la codificación de las instrucciones necesita muchos bits y, por este motivo, se habla de **arquitecturas VLIW** (del inglés *very long instruction word*).

Por otra parte, esta solución de múltiples ALU permite avanzar la ejecución de una serie de instrucciones que sean independientes de otra que ocupe una unidad de cálculo unos cuantos ciclos. Es el caso de las unidades de coma flotante, que necesitan mucho tiempo para acabar una operación en comparación con una ALU que trabaje con números enteros.

Otra opción para aumentar el rendimiento es disponer de varias unidades de procesamiento independientes (o *cores*, en inglés), cada una con su propio *pipeline*, si es el caso.

Esta microarquitectura *multi-core* es adecuada para procesadores que ejecutan varios programas en paralelo, ya que en un momento determinado cada *core* se puede ocupar de ejecutar un programa diferente.

### 3.3.4. Microarquitecturas con CPU y memoria diferenciadas

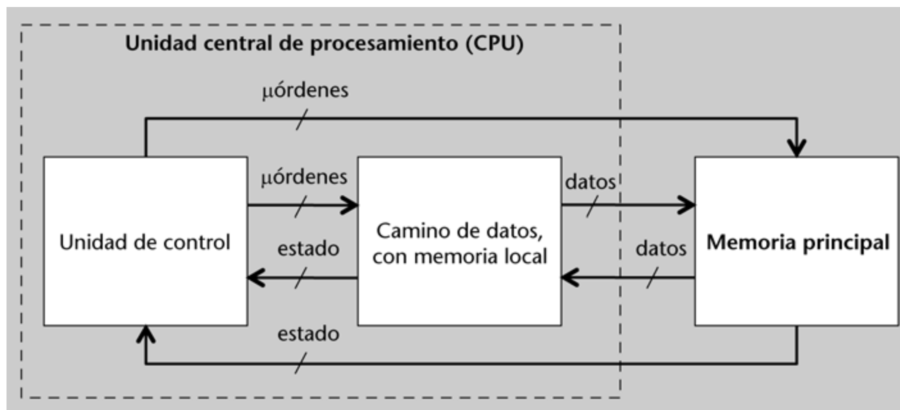
Dada la necesidad de disponer de procesadores con memorias de gran capacidad, se construyen de manera que la parte procesadora tenga recursos de memoria suficientes para trabajar de un modo eficiente (es decir, que el número de instrucciones ejecutadas por unidad de tiempo sea el mayor posible), pero que no incluya el grueso de la información (datos e instrucciones), ya que se puede almacenar mejor con tecnología específica.

Así, el modelo de construcción de los procesadores debe tener en cuenta los condicionantes tecnológicos que existen de cara a su materialización. Generalmente, las memorias se construyen con tecnologías que incrementan su capacidad. Los circuitos de procesamiento, en cambio, se hacen con unos criterios muy diferentes: poder procesar muchos datos en el menor tiempo posible, por lo que la tecnología es distina. Esto provoca que las microarquitecturas de los procesadores deban considerar que la unidad de procesamiento habrá de estar dividida en dos partes, por motivos tecnológicos:

- La **unidad central de procesamiento**, o **CPU** (de las siglas en inglés), es la parte del procesador que se dedica a realizar el procesamiento de la información. Se incluye también la unidad de control correspondiente.

- La **memoria principal** es la parte del procesador que se ocupa de almacenar datos e instrucciones (programas). Como se ha visto, puede estar organizada de manera que se vea como un único bloque (arquitectura de Von Neumann) o como dos bloques que guardan datos e instrucciones por separado (arquitectura de Hardvard).

Figura 56. Organización de un procesador con CPU y memoria




La relación entre las CPU y las memorias tiene mucha influencia en el rendimiento del procesador. Generalmente, las CPU incluyen un banco de registros lo suficientemente grande como para contener la mayoría de los datos que se emplean en un momento determinado, sin la necesidad de acceder a memoria, con independencia de si se implementa un RISC o un CISC. En todo caso, por grande que sea este banco de registros, no puede contener los datos necesarios para la ejecución de un programa cualquiera y se debe recurrir a las memorias externas a la CPU.

Las **arquitecturas *load/store*** se pueden construir sobre microarquitecturas optimizadas para realizar operaciones simultáneas de acceso a memoria y operaciones internas en la CPU. Normalmente, son construcciones en las que se puede ejecutar en paralelo una instrucción *load* o *store* con alguna otra de trabajo sobre registros de la CPU. En el caso contrario, con ISA que incluyen instrucciones de todo tipo con acceso a memoria y operaciones, es más complicado de llevar a cabo ejecuciones en paralelo, ya que las propias instrucciones imponen una secuencia en la ejecución de los accesos a memoria y la realización de las operaciones.

Sin embargo, hay microarquitecturas que separan la parte de trabajo con datos, de la parte de trabajo con direcciones de memoria para poder materializar CPU más rápidas.

De manera similar, también es habitual para determinados procesadores separar la memoria de datos de la de instrucciones. De este modo se puede conseguir un grado de paralelismo elevado y una mejora del rendimiento en cuanto a número de instrucciones ejecutadas por unidad de tiempo (a veces, se habla de los MIPS o millones de instrucciones por segundo que puede ejecutar un determinado procesador).

En cualquier caso, para un procesador de propósito general suele ser más conveniente sacrificar este factor de mejora del rendimiento para poder ejecutar una variedad mayor de programas.

Sea cual sea la microarquitectura del procesador, las unidades de procesamiento acaban teniendo una velocidad de trabajo más elevada que la de las memorias de gran capacidad. Eso sucede, en parte, porque uno de los factores que más pesa a la hora de materializar una memoria es, precisamente, la capacidad por encima de la velocidad de trabajo. Ahora bien, sabiendo que los programas presentan mucha localidad de todo tipo, es posible tener los datos y las instrucciones más utilizadas (o las que se puedan utilizar con más probabilidad) en memorias más rápidas, aunque no tengan tanta capacidad. Estas memorias son las denominadas **memorias cachés** (porque son transparentes a la unidad de procesamiento).

**Memorias caché**

Las **memorias caché** se denominan así porque son memorias que se emplean como escondites para almacenar cosas fuera de la vista de los demás. En la relación entre las CPU y las memorias principales se ponen memorias cachés, más rápidas que las principales, para almacenar datos e instrucciones de la memoria principal para que la CPU, que trabaja a mayor velocidad, tenga un acceso más rápido a ellas. Si no hubiera memorias cachés, la transferencia de información se haría igualmente, pero a la velocidad de la memoria principal. De aquí que sean como escondites o escondrijos.

### 3.3.5. Procesadores de propósito general

Los **procesadores de propósito general** (o **GPP**, del inglés *general-purpose processor*) son procesadores destinados a poder ejecutar programas de todo tipo para aplicaciones de procesamiento de información. Es decir, son procesadores para ordenadores de uso genérico, que pueden servir tanto para gestionar una base de datos en una aplicación profesional como para jugar.

Por este motivo, suelen ser procesadores con un repertorio de instrucciones suficientemente amplio para ejecutar con eficiencia un programa que necesita realizar muchos cálculos u otro con un requerimiento más elevado de movimiento de datos. Aunque este hecho hace pensar en una arquitectura CISC, es frecuente encontrar GPP construidos con RISC porque suelen tener ciclos de ejecución de instrucciones más cortos que mejoran el rendimiento de las microarquitecturas en *pipeline*. Hay que tener en cuenta que los *pipeline* con muchas etapas son difíciles de gestionar cuando, por ejemplo, hay saltos.

Dada la variabilidad de volúmenes de datos y de tamaño de programas que puede haber en un caso general, los GPP están construidos siguiendo los principios de la arquitectura de Von Neumann, en la que código y datos residen en un único espacio de memoria.

### 3.3.6. Procesadores de propósito específico

En el caso de los procesadores destinados a algún tipo de aplicación concreto, tanto el repertorio de instrucciones como la microarquitectura se ajustan para obtener la máxima eficiencia posible en la ejecución de los programas correspondientes.

El repertorio de instrucciones incluye algunas que son específicas para el tipo de aplicación al que se destinará el procesador. Por ejemplo, puede tener operaciones con vectores o con coma flotante, si se trata de aplicaciones que necesitan mucho procesamiento numérico, como la generación de imágenes o

el análisis de vídeo. Evidentemente, si este es el caso, la microarquitectura correspondiente debe tener en cuenta que la CPU deberá utilizar varias unidades aritmeticológicas (o de otros elementos de procesamiento) en paralelo.

De hecho, el ejemplo anterior se corresponde con los **procesadores digitales de señal** o **DSP** (del inglés, *digital signal processors*). Por lo tanto, los DSP están orientados a aplicaciones que requieran el procesamiento de un flujo continuo de datos que se puede ver como una señal digital que hay que ir procesando para obtener una de salida convenientemente elaborada.

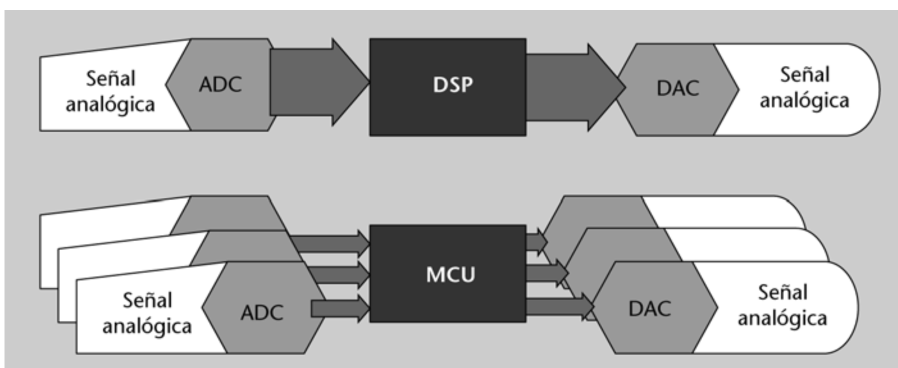
Para el caso particular de las imágenes, hay DSP específicos, denominados **unidades de procesamiento de imágenes** o **GPU** (del inglés, *graphics processing units*) que tienen una ISA similar, pero que suelen utilizar un sistema de memoria diferente con las direcciones organizadas en 2D para facilitar las operaciones con las imágenes.

Un caso de características muy diferentes es el de los controladores, que deben captar muchas señales de los sistemas que controlan y generar las señales de control convenientes. Las aplicaciones de control, pues, trabajan con muchas señales que, si bien finalmente también son señales digitales, no ocupan demasiados bits y su procesamiento individual no es especialmente crítico. Ahora bien, estas aplicaciones necesitan procesadores con un repertorio de instrucciones que incluya una amplia variedad de operaciones de entrada/salida y una microarquitectura que las permita ejecutar de manera eficiente.

En estos casos, los procesadores actúan como **microcontroladores** o **MCU** (del inglés, *microcontroller unit*). Por el hecho de tener capacidad de captar datos del exterior y sacar hacia fuera sin necesidad de elementos adicionales, se pueden ver como pequeños computadores: periféricos y procesador en un único chip.

La dualidad entre DSP y MCU se puede resumir en el hecho de que los primeros procesan pocas señales de muchos bits muy rápidamente y de que los segundos son capaces de tratar muchas señales de pocos bits casi simultáneamente.

Figura 57. Ejemplificación de la dualidad entre DSP y MCU (ADC y DAC son, respectivamente, convertidores analógico-digital y digital-analógico)



En la práctica hay muchas aplicaciones que necesitan combinar las características de los DSP y de los MCU, lo que lleva a muchos procesadores específicos a conocerse de un modo u otro según las características dominantes, pero no porque las otras no las tengan.

Además de implementar un repertorio específico de instrucciones, existen dos factores que hay que tener en cuenta: el tiempo que se tarda en ejecutarlas y el consumo de energía para hacerlo. A menudo hay aplicaciones que son muy exigentes en los dos aspectos.

> Por ejemplo, cualquier dispositivo móvil con capacidad de captar/mostrar vídeo necesita una capacidad de procesamiento muy elevada en un tiempo relativamente corto y sin consumir demasiada energía.

El tiempo y el consumo de energía son factores contradictorios y las microarquitecturas orientadas a la rapidez de procesamiento suelen consumir mucha más energía que las que incluyen mecanismos de ahorro energético a cambio de ir un poco más lentos.

Para ir rápido, los procesadores incluyen mecanismos de *pipelining* y de procesamiento paralelo. Además, suelen estar construidos según la arquitectura Hardvard para trabajar de manera concurrente con instrucciones y datos. Para consumir menos, incluyen mecanismos de gestión de la memoria interna de la CPU que evite lecturas y escrituras innecesarias a la memoria principal. Sin embargo, los procesadores de bajo consumo (*low-power processors*) suelen ser más sencillos que los de alto rendimiento (*high-performance processors*) porque la gestión de los recursos se complica cuando hay muchos.

## 3.4. Computadores

Los **computadores** son máquinas que incluyen, al menos, un procesador para poder procesar información, en el sentido más general. Habitualmente trabajan de manera interactiva con los usuarios, aunque también pueden trabajar independientemente. La interacción de los computadores no se limita a los usuarios, sino que también abarca el entorno o el sistema en el que se encuentran. Así, hay computadores que se perciben como tales y que tienen un modo de trabajo interactivo con los humanos: con ellos jugamos, creamos, consultamos y gestionamos información de todo tipo, trabajamos y hacemos casi cualquier cosa que se nos pase por la imaginación. También hay algunos más "escondidos", que controlan aparatos de cualquier clase o que permiten que haya dispositivos de todo tipo con un "comportamiento inteligente".

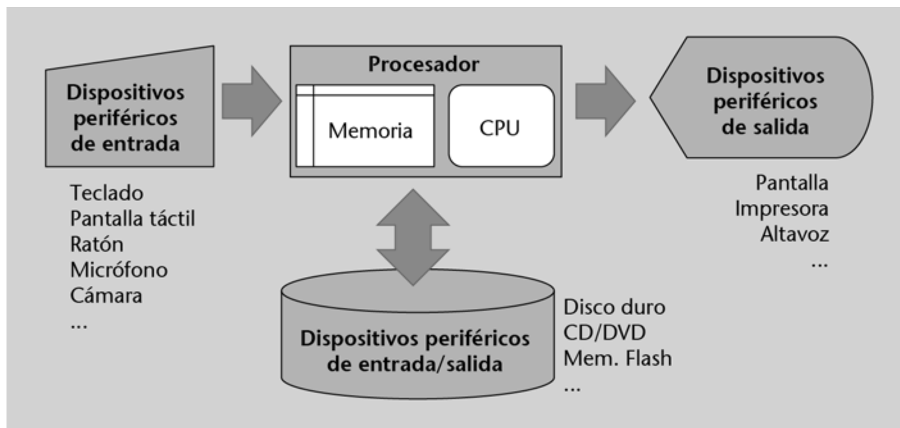
Sean como sean, los computadores tienen unos rasgos comunes que se comentarán en el resto de la sección.

### 3.4.1. Arquitectura básica

Los computadores procesan información. Para hacerlo, necesitan un procesador. Los procesadores son los núcleos de los computadores.

Para que los procesadores puedan hacer su trabajo, hay que darles información que procesar (y también los programas para hacerlo) y deben estar dotados de mecanismos para recoger la información procesada. De estas tareas se ocupan los denominados **dispositivos periféricos** o, simplemente, **periféricos** (y son "periféricos" porque se encuentran en la periferia del núcleo del computador).

Figura 58. Arquitectura general de un computador




Los periféricos de entrada más habituales son el teclado y el ratón; y los de salida más comunes, la pantalla (o monitor) y la impresora. Algunos habréis notado que hay bastantes más periféricos: altavoces, escáneres (para capturar imágenes impresas), micrófonos, cámaras, televisores y una retahíla adicional que se amplía continuamente para ajustarse a las aplicaciones informáticas que puede llevar a cabo un computador.

También existen periféricos que pueden realizar las dos tareas. Es decir, pueden ocuparse de la entrada de los datos en el ordenador y de la salida de los datos resultantes. Los dispositivos periféricos de entrada/salida más visibles son las unidades (los elementos del ordenador) de discos ópticos (CD, DVD, Blu-ray, etc.) y de tarjetas de memoria. Otros periféricos de este tipo son los discos duros que están dentro de la caja del ordenador y los módulos de conexión inalámbrica a red.

Generalmente, este tipo de periféricos se utiliza para almacenar datos que se pueden recuperar cuando un determinado proceso lo solicite e, igualmente, modificarlos si se requiere. El caso de los módulos de conexión a red resulta un poco especial, dado que son unos periféricos que no almacenan información pero que permiten obtenerla y enviarla a través de la red.

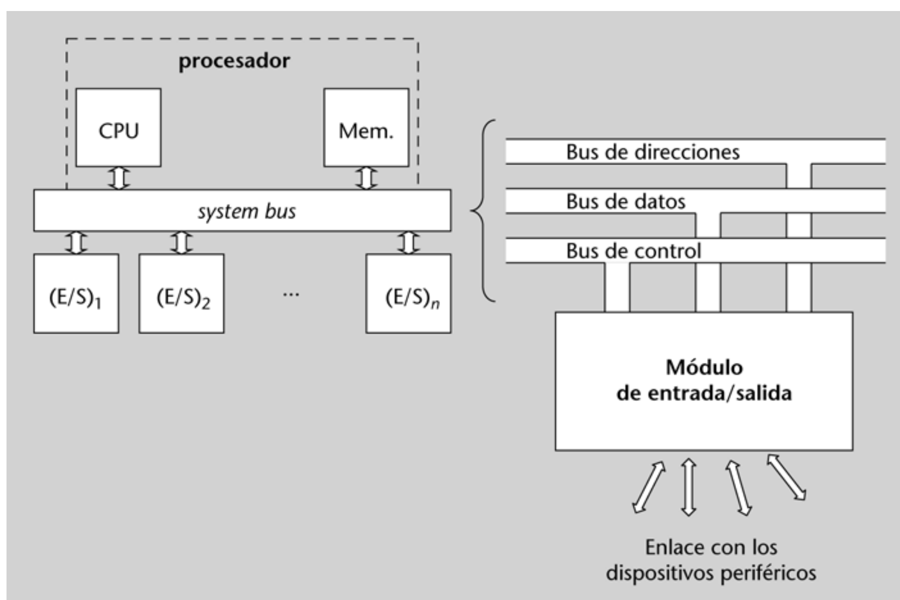
Para comunicarse con el procesador, los dispositivos periféricos disponen de controladores que, además de gobernar su funcionamiento, se relacionan con un **módulo de entrada/salida** del computador correspondiente. Estos módulos hacen de puente entre el periférico y el procesador, adaptando las diferentes maneras de funcionar, los formatos de los datos y la velocidad de trabajo. Por ejemplo, las lecturas de disco se realizan por bloques de varios kB que se

procesan para eliminar los errores y organizarlos convenientemente para poder ser transferidos al procesador en unidades (de pocos bytes) y con velocidades de transferencia ajustadas a sus buses.

A modo de ejemplo, se puede comentar que la CommUnit del YASP realiza las funciones de módulo de entrada/salida del procesador. Cualquier periférico que se conecte a él recibe y envía señales de este módulo, que, a su vez, se relaciona tanto con la unidad de control como con el registro A si se trata de una transferencia de datos. Evidentemente, un procesador más complejo necesita más módulos de E/S, con funcionalidades específicas para cada tipo de periférico, que son también más complejos.

Por lo tanto, una arquitectura básica de un computador debería tener varios módulos de entrada/salida conectados a un bus del sistema, que también tendría conectados la CPU y la memoria principal, tal y como se muestra en la figura 59.

Figura 59. Esquema de un computador basado en un único bus




El bus del sistema se organiza en tres buses diferentes:

**1)** El bus de datos se utiliza para transportar datos entre todos los elementos que conecta. Frecuentemente, los datos van de la memoria principal a la CPU, pero también al revés. Las transferencias entre CPU y módulos de E/S o entre módulos de E/S y memoria son relativamente más esporádicas.

**2)** El bus de direcciones transmite las direcciones de los datos (y de las instrucciones) a las que accede la CPU, bien sea a la memoria, bien a algún módulo de E/S. También lo aprovechan los módulos de E/S para acceder directamente a datos en memoria.

**3)** El bus de control transmite todas las señales de control para que todos los elementos puedan comunicarse correctamente. Desde la CPU la unidad de control es la encargada de recibir las señales que le tocan y emitir las correspondientes en cada estado en el que se encuentre. Es importante tener presen-

te que, de hecho, cada módulo conectado al bus tiene su propia unidad de control.

El acceso a los periféricos desde el procesador se realiza mediante módulos de entrada/salida específicos. En una arquitectura básica, pero perfectamente funcional, los módulos de entrada/salida se pueden relacionar directamente con la CPU, ocupando una parte del espacio de direcciones de la memoria. Es decir, existen posiciones de la memoria del sistema que no se corresponden con datos almacenados en la memoria principal, sino que son posiciones de las memorias de los módulos de E/S. De hecho, estos módulos tienen una memoria interna en la que recoger datos para el periférico o en la que almacenar temporalmente los que provienen de este.

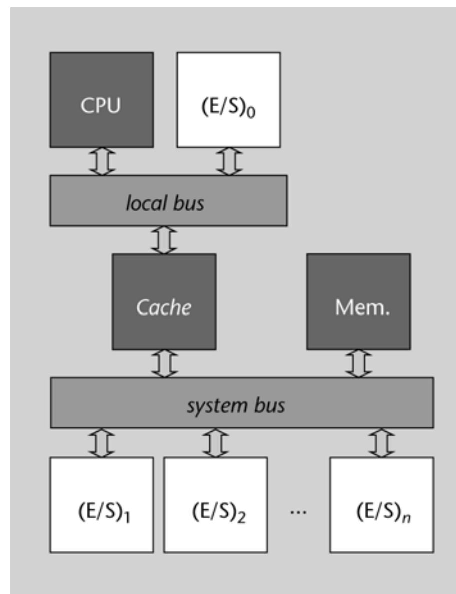
Las operaciones de entrada/salida de datos programadas implicarían sincronizar el procesador con el periférico correspondiente. Es decir, que el procesador debería esperar la disponibilidad del periférico para hacer la transmisión de los datos, lo que significaría una pérdida notable de rendimiento en la ejecución de la aplicación asociada. Para entenderlo, debemos considerar que un disco puede transferir datos a velocidades comparables (en bytes/segundo) a las que puede trabajar una CPU o la memoria principal, pero la parte mecánica provoca que el acceso a los datos pueda tardar unos cuantos milisegundos, un tiempo que sería perdido por el procesador.

En general, la transferencia de datos entre periféricos y procesador se realiza mediante módulos con mecanismos de **acceso directo a la memoria** (**DMA**, del inglés *direct memory access*). Los módulos dotados de DMA transfieren datos entre su memoria interna y la principal del computador sin intervención de la CPU. De esta manera, la CPU queda “liberada” para ir ejecutando instrucciones del programa en curso. En la mayoría de los casos, los módulos con DMA y la CPU se comunican para establecer cuál de ellos tiene acceso a la memoria mediante el bus del sistema, con prioridad para la CPU, obviamente.

Como la memoria principal todavía resulta relativamente lenta en cuanto a la CPU, es habitual interponer memoria caché. Como ya se ha comentado, esta memoria resulta transparente a la CPU y a la memoria principal: el controlador del módulo de la memoria caché se ocupa de “capturar” las peticiones de lectura y escritura de la CPU y de responder como si fuera la memoria principal, pero más rápidamente. En la figura 60, se muestra la organización de un computador con un bus local para comunicar la CPU con la memoria caché y un módulo de entrada/salida para las comunicaciones con los periféricos que no se realizan mediante la memoria. El bus local trabaja a más velocidad que el de sistema, más de acuerdo con las frecuencias de operación de la CPU y de la memoria caché.

Esta configuración básica de un computador se puede mejorar incorporando más recursos. En el subapartado siguiente se comentarán brevemente algunas ampliaciones posibles, según la aplicación a la que se destinen los computadores correspondientes.

Figura 60. Esquema de un computador con memoria caché




### 3.4.2. Arquitecturas orientadas a aplicaciones específicas

Las arquitecturas genéricas pueden extenderse con recursos que incidan en la mejora de algún aspecto del rendimiento de una aplicación específica: si necesita mucha capacidad de memoria, se las dota con más recursos de memoria, y si necesita mucha capacidad de procesamiento, se las dota con más recursos de cálculo.

La capacidad de procesamiento se puede aumentar con coprocesadores especializados o multiplicando el número de procesadores del computador.

> Por ejemplo, muchos ordenadores incluyen GPU como coprocesadores que liberan a las CPU del trabajo de tratamiento de imágenes. En este caso, cada unidad (la CPU y la GPU) trabaja con su propia memoria. En el caso de múltiples procesadores, la memoria principal suele ser compartida, pero cada CPU tiene su propia memoria caché.

El aumento de la capacidad de la memoria se puede realizar con la contrapartida de mantener (o aumentar) la diferencia de velocidades de trabajo de memoria principal y CPU. La solución pasa por interponer más de un nivel de memorias cachés: las más rápidas y pequeñas cerca de la CPU y las más lentas y grandes cerca de la memoria principal.

**Computador de alto rendimiento**

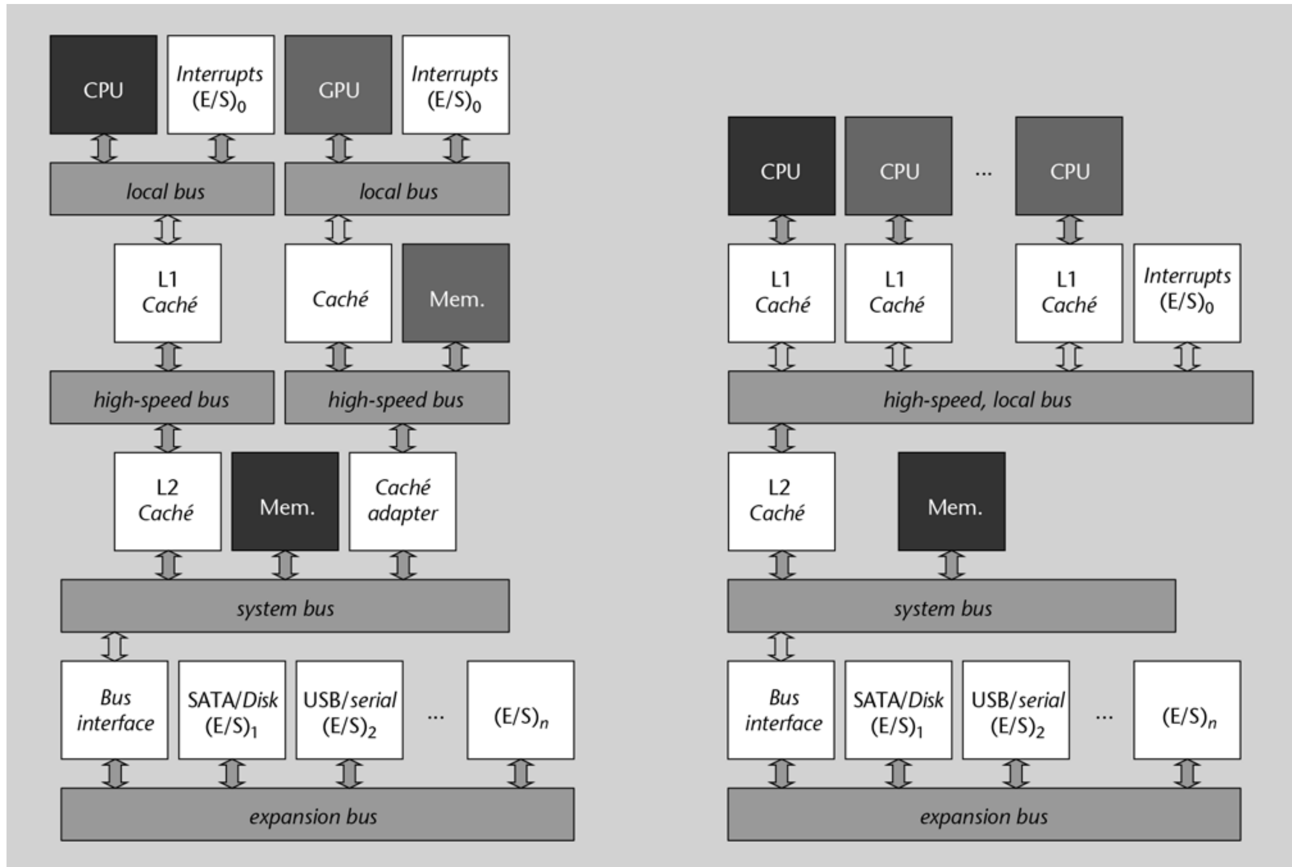
Un **computador de alto rendimiento** es aquel capaz de ejecutar en un mismo tiempo programas más grandes en memoria y número de instrucciones que los normales, entendiendo por normales los que son mayoría en un momento determinado.

Las crecientes necesidades de procesamiento y memoria en todo tipo de aplicaciones han provocado que las arquitecturas originalmente concebidas para computadores de alto rendimiento sean cada vez más comunes en cualquier ordenador.

En la figura 61 se muestran dos arquitecturas específicas para aplicaciones gráficas y multiproceso. Las dos utilizan dos niveles de memoria caché: caché L1 y L2, y organizan el sistema en una jerarquía de buses. En la dedicada a apli-

caciones gráficas, hay un coprocesador específico (una GPU) que tiene una memoria principal local y una memoria caché propias. La opción multiprocesadora se basa en varias CPU que tienen una memoria caché local, pero que comparten una general para la memoria principal.

Figura 61. Arquitecturas de computadores con GPU (izquierda) y múltiples CPU (derecha)




Estas dos configuraciones se pueden combinar para obtener computadores de muy alto rendimiento para todo tipo de aplicaciones y, especialmente, para aquellas intensivas en cálculo.

La evolución tecnológica posibilita integrar cada vez más componentes en un único chip, lo que aporta la ventaja de poner más recursos para construir los computadores, pero lo cual también complica cada vez más su arquitectura.

Por ejemplo, hay chips *multi-core* que incluyen varios niveles de memoria caché y también coprocesadores específicos de tipo GPU.

## Resumen

Los computadores son circuitos secuenciales complejos, muy complejos. Se ha visto que están organizados en varios módulos más pequeños que interactúan entre ellos para conseguir ejecutar programas descritos en lenguajes de alto nivel de la manera más eficiente posible.

Un computador se puede despojar de sus dispositivos periféricos y continuar manteniendo la esencia que lo define: la capacidad de procesar información de manera automática siguiendo un programa. Sin embargo, los procesadores también pueden estar organizados de maneras muy diferentes, aunque, para mantener la capacidad de ejecutar cualquier programa, siempre trabajan ejecutando una secuencia de instrucciones almacenada en memoria. De hecho, esta organización de la máquina se conoce como **arquitectura de Von Neumann**, en honor al modelo establecido por un computador descrito por este autor.

Se ha visto que los procesadores pueden materializarse a partir de descripciones algorítmicas de máquinas que interpretan las instrucciones de un determinado repertorio y también que esto se puede hacer tanto para construir unidades de control sofisticadas como para materializar procesadores sencillos.

Las máquinas algorítmicas también se pueden materializar exclusivamente como circuitos secuenciales. En este sentido, se ha tratado un caso particular a modo de ejemplo. Estas máquinas son, en el fondo, máquinas de estados algorítmicas focalizadas más en los cálculos que en las combinaciones de señales de entrada y de salida. Se ha visto que las ASM son útiles cuando las transiciones entre estados dependen de pocas señales binarias (externas o internas) y que resultan efectivas para la implementación de controladores de todo tipo en estas circunstancias.

Con todo, el problema del aumento del número de estados cuando hay secuencias de acciones (y/o cálculos) en una máquina también se resuelve bien con una PSM. En estas máquinas, algunos estados pueden representar la ejecución de un programa completo y, por lo tanto, tienen menos estados que una máquina de estados con los programas desplegados como secuencias de estados. A cambio, deben utilizar una variable específica, que se denomina **contador de programa**.

Se ha visto también que todas estas máquinas, en general, se pueden materializar de acuerdo con una arquitectura de máquina de estados con camino de datos (FSMD) y que, de hecho, son casos concretos de máquinas de estados finitos extendidas.

Las EFSM son un tipo de máquinas que incluyen la parte de control y de procesamiento de datos en la misma representación, lo que lo facilita tanto el análisis como la síntesis.

Se ha tratado un caso paradigmático de las EFSM, el contador. El contador utiliza una variable para almacenar la cuenta y, con ello, discrimina los estados de la máquina de control principal del estado general de la máquina, que incluye también el contenido de todas las variables que contiene.

Previamente, se ha visto que las máquinas de estados finitos o FSM sirven para modelar controladores de todo tipo, tanto de sistemas externos como de la parte de procesamiento de datos del circuito.

En resumen, se ha pasado de un modelo de FSM con entradas y salidas de un bit que servía para controlar "cosas" a modelos más completos que, a la vez que permitían representar comportamientos más complejos, admitían la expresión de algoritmos de interpretación de programas. Se han mostrado ejemplos de materialización de todos los casos para facilitar la comprensión del funcionamiento de los circuitos secuenciales más sencillos y también la de aquellos que son más complejos de lo que puede abarcar este material: los que constituyen las máquinas que denominamos **computadores**.

# Ejercicios de autoevaluación

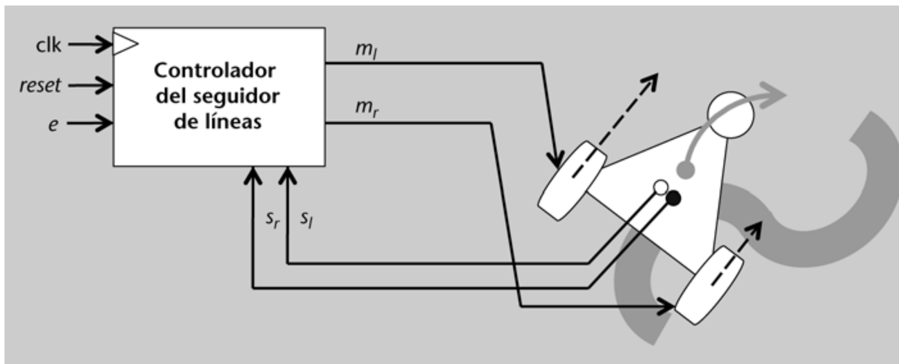
Los problemas que se proponen a continuación se deberían resolver una vez acabado el estudio del módulo, de manera que sirvan para comprobar el grado de consecución de los objetivos indicados al principio. No obstante, los cuatro primeros se pueden aprovechar como preparación de una posible práctica de la asignatura y, por este motivo, sería útil resolverlos en paralelo al estudio de la última parte del módulo (apartado 3).

**1.** Este ejercicio debe servir para reforzar la visión de las FSM como representaciones del comportamiento de controladores. Para ello, se pide que diseñéis el controlador de un robot seguidor de líneas. El vehículo del robot se mueve por par diferencial (figura 17), tal y como se explica en el apartado 1.3, y está dotado de un par de sensores en la parte de abajo, que sirven para detectar si el robot está encima de la línea total o parcialmente. Por lo tanto, las entradas del controlador son las provenientes de estos sensores, $s_l$ y $s_r$, por sensor izquierdo y derecho, respectivamente. Hay una entrada adicional de activación y parada, *e*, que debe estar a 1 para que el robot siga la línea. Las salidas son las que controlan los motores izquierdo y derecho para hacer girar las ruedas correspondientes: $m_l$ y $m_r$. El funcionamiento es el que se muestra en la tabla siguiente:

| $m_l$ | $m_r$ | Efecto |
|---|---|---|
| 0 | 0 | Robot parado |
| 0 | 1 | Giro a la izquierda |
| 1 | 0 | Giro a la derecha |
| 1 | 1 | Avance recto adelante |

Los giros se deben hacer cuando, al seguir una línea, uno de los dos sensores no la detecte. Por ejemplo, si el sensor izquierdo la deja de detectar, se ha de girar hacia la derecha.

Figura 62. Esquema de bloques del robot seguidor de líneas




Se supone que, inicialmente, el robot se hallará en una línea y que, en el caso de que no la detecte ningún sensor, se detendrá. Con esta información, se debe elaborar el diagrama de la FSM correspondiente. (No hay que hacer el diseño del circuito que lo implemente.)

**2.** En este problema se repasa el hecho de que las EFSM son FSM con operandos (habitualmente, números) y operadores de múltiples bits y que se construyen con una arquitectura de FSMD: debéis construir la EFSM de un detector de sentido de avance de un eje e implementar el circuito correspondiente.

Este detector utiliza lo que se denomina un "codificador de rotación", que es un dispositivo que convierte la posición angular de un eje en un código binario. El detector que se ha de construir tendrá, como entrada, un número natural de tres bits, $A = (a_2, a_1, a_0)$, que indicará la posición angular absoluta del eje, y, como salida, dos bits, $S = (s_1, s_0)$, que indicarán si el eje gira en sentido del reloj, $S = (0, 1)$, o si lo hace en sentido contrario, $S = (1, 0)$. Si el eje está parado o no cambia de sector, la salida será (0, 0). Un sector queda definido por un rango de posiciones angulares que tienen el mismo código.

Aunque no se represente en la EFSM del detector, la materialización del circuito correspondiente debe tener una entrada de *reset* que fuerce la transición hacia el estado inicial.

La figura 63 presenta el sistema que hay que diseñar. El detector utiliza un codificador de rotación que proporciona la posición del ángulo en formato de número natural de 3 bits según la codificación que se muestra en la tabla siguiente:

| Sector | $s_2$ | $s_1$ | $s_0$ |
|---|---|---|---|
| [0°, 45°) | 0 | 0 | 0 |
| [45°, 90°) | 0 | 0 | 1 |
| [90°, 135°) | 0 | 1 | 0 |
| [135°, 180°) | 0 | 1 | 1 |
| [180°, 225°) | 1 | 0 | 0 |
| [225°, 270°) | 1 | 0 | 1 |
| [270°, 315°) | 1 | 1 | 0 |
| [315°, 360°) | 1 | 1 | 1 |

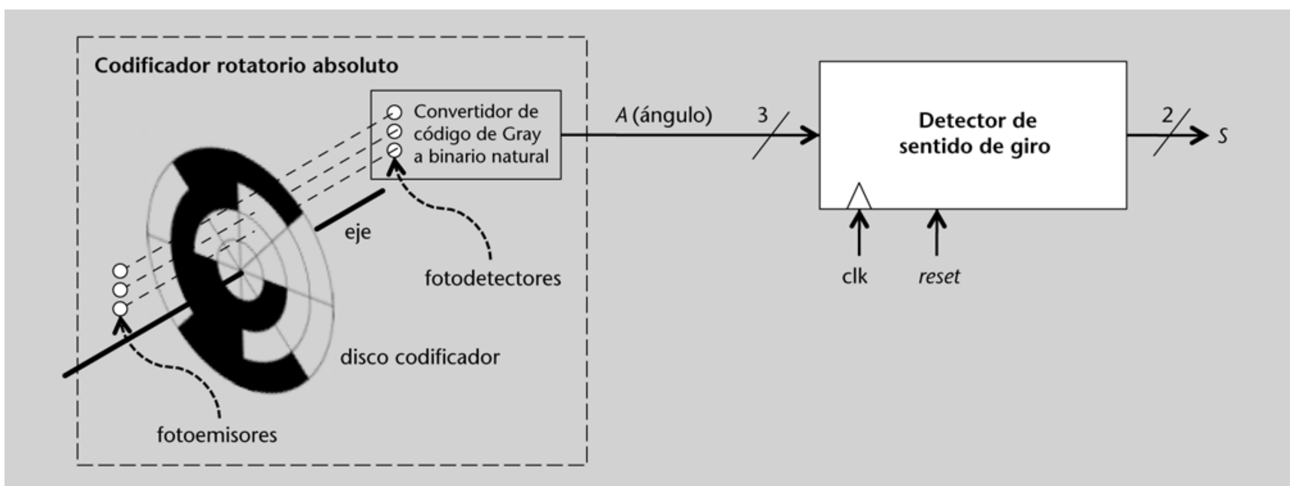
**Códigos de Gray**

Los códigos de Gray se utilizan en este tipo de codificadores rotatorios porque entre dos códigos consecutivos solo cambia un bit, lo que hace que el cambio de sector se detecte con este cambio. Entre el código del sector inicial y el del final de este cambio se pueden detectar, transitoriamente, cambios en más de un bit. En este caso, se descartan los códigos intermedios.

Hay que tener presente que del sector [315°, 360°) se pasa directamente al sector [0°, 45°), al girar en el sentido de las agujas del reloj.

El codificador rotatorio que se presenta funciona con un disco con tres pistas (una por bit) que está dividido en 8 sectores. A cada sector le corresponde un código de Gray de tres bits que se lee con tres fotodetectores y que, posteriormente, se convierte en un número binario natural que identifica el sector tal y como se ha presentado en la tabla anterior.

Figura 63. Detector de sentido de giro con un codificador rotatorio absoluto



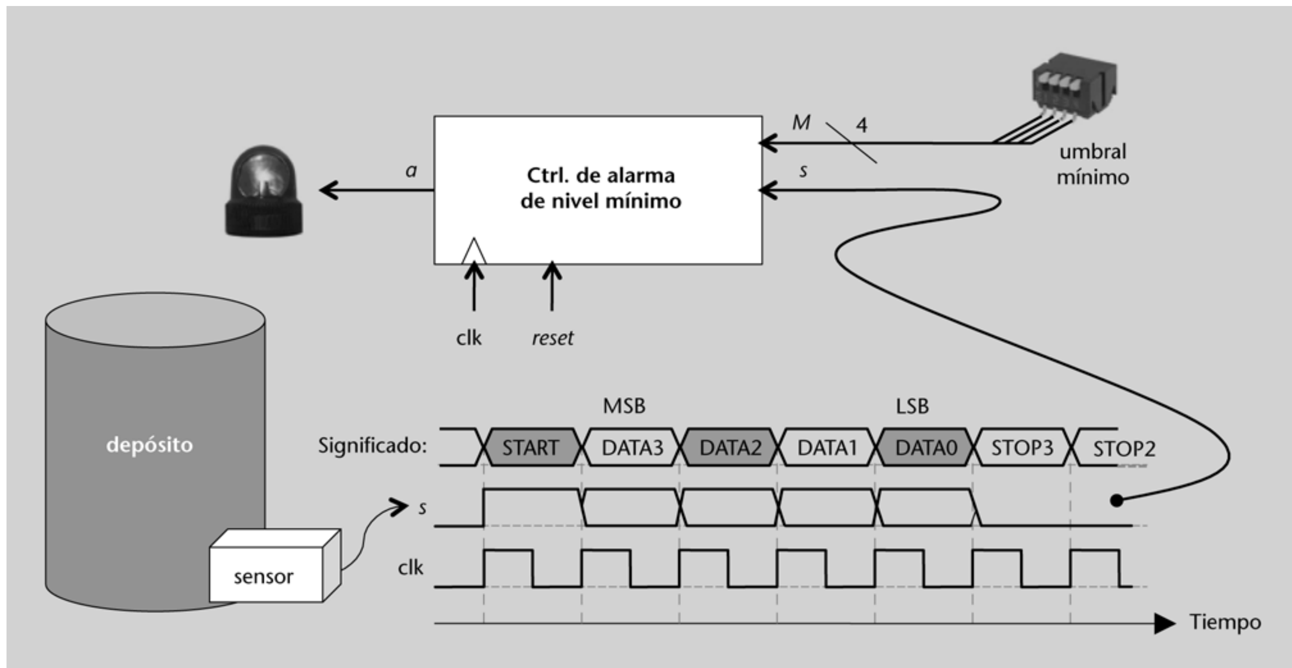

Para resolver este problema hay que tener presente que el detector de sentido de giro debe comparar el sector actual con el anterior. Se supone que la cadencia de las lecturas de los ángulos *A* es suficientemente elevada como para que no pueda haber saltos de dos o más sectores entre dos lecturas consecutivas.

El circuito correspondiente se puede diseñar utilizando los recursos que os convengan de los que habéis visto hasta ahora.

**3.** Para ver la utilidad de las PSM a la hora de representar comportamientos que incluyen secuencias de acciones, debéis elaborar el modelo de un controlador de una alarma de nivel mínimo de un depósito.

El controlador enviará, por la señal de salida *a*, un 1 durante un ciclo de reloj a una alarma para activarla en caso de que el nivel del depósito sea igual o menor que una referencia, *M*, dada. La referencia se establece con unos pequeños conmutadores. El nivel del líquido en el depósito le proporciona un sensor a modo de número binario natural de 4 bits. Este número indica el porcentaje de llenado del depósito, desde el 0% ($0000_{(2}$) hasta el 100% ($1111_{(2}$) y, por lo tanto, cada unidad equivale al 6,25%. Con el fin de reducir cables, los datos del sensor se envían en series de bits. Tal y como se muestra en la figura 64, primero se envía un bit en 1 y luego los 4 bits que conforman el número que representa el porcentaje de llenado del depósito. Los bits de este número se envían empezando por el más significativo y acaban por el menos significativo. Siempre habrá, como mínimo, 4 bits a cero siguiendo el último bit del número en una transmisión.

Figura 64. Controlador de alarma de nivel mínimo




Diseñad la PSM que representa el comportamiento de este controlador, teniendo presente que la alarma se debe mantener activada siempre que se cumpla que la lectura del sensor sea inferior o igual al umbral mínimo establecido y que el sensor envía datos cada vez que detecta algún cambio significativo en el nivel.

**4.** Este problema trata del uso de ASM en los casos en los que hay un número abundante de entradas, de manera que la representación del comportamiento sea más compacto e inteligible: hay que realizar el modelo de control de un horno de microondas.

El horno tiene una serie de sensores que le proporcionan información por medio de las señales que se listan a continuación:

- *s* indica que se ha apretado el botón de inicio/parada con un pulso a 1 durante un ciclo de reloj;
- *d* es 0 si la puerta está abierta, o 1, si está cerrada;
- *W* es un número relacionado con la potencia de trabajo del microondas, que va de los valores 1 a 4, codificados de 0 a 3, y
- *t* es una señal que se mantiene a 1 mientras el temporizador está en marcha: la rueda del temporizador se programa girándola en el sentido de las agujas del reloj y, a medida que pasa el tiempo, gira en sentido contrario hasta la posición inicial. La señal *t* solo está en cero cuando la rueda del temporizador está en la posición 0.

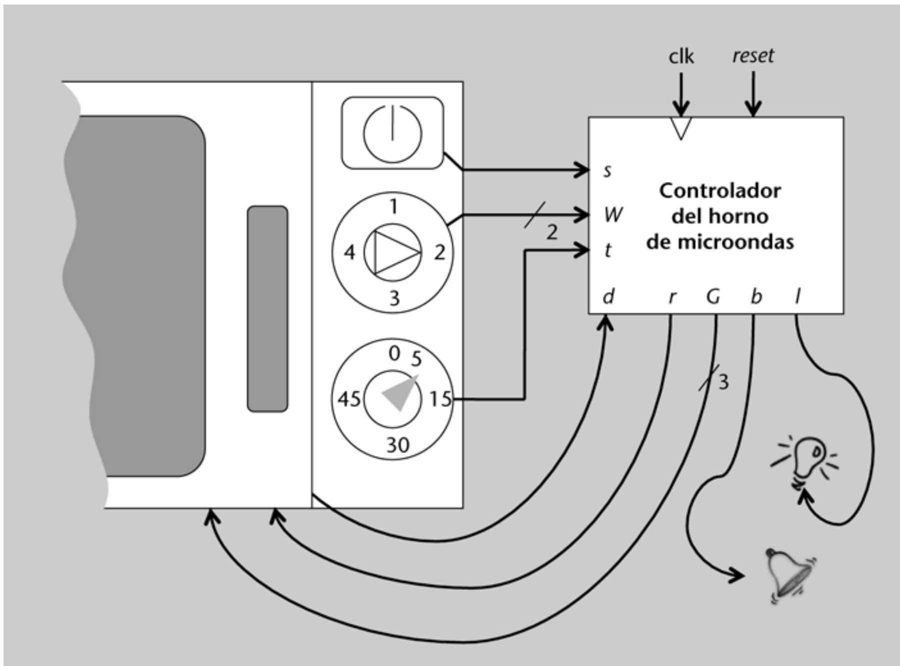
Hay que tener presente que el botón de inicio/parada se puede apretar en cualquier momento y que la rueda del temporizador solo puede volver a la posición 0 cuando haya pasado el tiempo correspondiente o si el usuario fuerza el retorno girándola hasta aquella posición. Si se aprieta para poner en marcha el horno con la puerta abierta o sin el temporizador activado, no tiene efecto y el horno continúa apagado. El horno se debe apagar en cualquier momento en el que se abra la puerta o se apriete el botón de inicio/parada.

El controlador toma la información proporcionada por estos sensores y realiza las actuaciones correspondientes, según el caso, mediante las señales siguientes:

- *b* se debe poner a 1 durante un ciclo de reloj para hacer sonar un timbre de aviso de fin del periodo de tiempo marcado por el temporizador;
- *l* controla la luz del interior, que solo está encendida si es 1 (como mínimo, durante el funcionamiento del horno);
- *r* debe ser 1 para hacer girar el plato interior, y
- *G* es un número natural que gobierna el funcionamiento del generador de microondas, siendo 0 el valor para apagarlo y 4 para funcionar a potencia máxima.

El esquema del conjunto se puede ver en la figura 65.

Figura 65. Esquema de bloques de un horno de microondas




Por lo tanto, se trata de que diseñéis la ASM correspondiente a un posible controlador del horno microondas que se ha descrito.

**5.** En muchos casos, los controladores deben realizar cálculos complejos que hay que tratar independientemente con esquemas de cálculo. En este ejercicio debéis diseñar uno de estos esquemas para calcular el resultado de la expresión siguiente:

$$ax^2 + bx + c$$

El esquema se debe organizar para utilizar los mínimos recursos posibles. Por simplicidad, todos los datos serán del mismo formato, tanto los de entrada como los intermedios y de salida.

**6.** Los diagramas de flujo sirven para representar algoritmos para hacer cálculos más complejos que pueden incluir esquemas como los anteriores. En este ejercicio se trata de ver cómo se puede transformar un diagrama de flujo en un circuito con arquitectura de FSMD. Para ello, implementad el cálculo de la raíz cuadrada entera.

La raíz cuadrada entera de un número $c$ es el valor entero $a$ que cumple la igualdad siguiente:

$$a^2 \leq c < (a + 1)^2$$

Es decir, la raíz cuadrada entera de un número $c$ es el valor entero más próximo por la izquierda a su raíz cuadrada real.

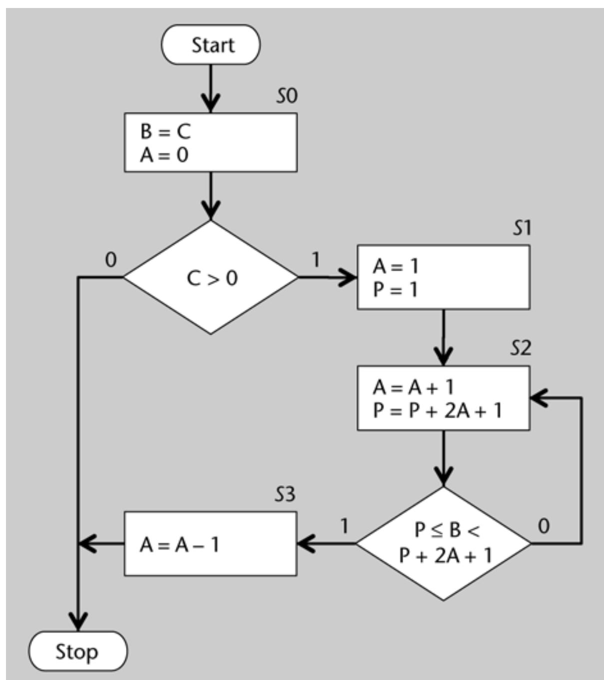
Para calcular $a$ se puede seguir un procedimiento iterativo a partir de $a = 1$, incrementando gradualmente $a$ hasta que cumpla la condición que se ha mencionado anteriormente.

En el diagrama de flujo de la figura 66 se puede ver el algoritmo correspondiente, con las variables en mayúscula, dado que son vectores de bits. Se incluye una comprobación de que $c$ no sea 0 porque el procedimiento iterativo fallaría en este caso. Para simplificar los cálculos de los cuadrados, se almacenan en una variable $P$ que se actualiza teniendo presente que $(A + 1)^2 = A^2 + 2A + 1$. Es decir, que el valor siguiente de la variable $P$ es $P + 2A + 1$.

A la vista del diagrama, explicad por qué es necesario el estado $S3$.

Diseñad el circuito correspondiente, siguiendo la arquitectura de FSMD, con unidad de control según el modelo visto que reproduce el flujo del diagrama.

Figura 66. Algoritmo para calcular la raíz cuadrada entera




**7.** Se trata de ver cómo se pueden construir procesadores a partir de otros ampliando el repertorio de instrucciones. En este caso, debéis modificar el diagrama de flujo de la figura 48, que representa el comportamiento del Femtoproc, de manera que pueda trabajar con más datos que los que se pueden almacenar en los 6 registros libres que tiene. Para ello, hay que tener presente que el nuevo repertorio tiene un formato de instrucciones de 9 bits, tal y como se muestra en la tabla siguiente.

| Instrucción | Bits | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | 0 | 0 | 0 | *operando*$_1$ | | | *operando*$_0$ | | |
| AND | 0 | 0 | 1 | *operando*$_1$ | | | *operando*$_0$ | | |
| NOT | 0 | 1 | 0 | *operando*$_1$ | | | *operando*$_0$ | | |
| JZ | 0 | 1 | 1 | dirección de salto | | | | | |
| LOAD | 1 | 0 | dirección de datos | | | | *operando*$_0$ | | |
| STORE | 1 | 1 | dirección de datos | | | | *operando*$_0$ | | |

De hecho, si el bit más significativo es 0, se sigue el formato anterior. Si es 1, se introducen las instrucciones de LOAD y STORE, que permiten leer un dato de la memoria de datos o escribirlo en esta, respectivamente. En estos casos, el *operando*$_0$ identifica el registro (Rf) del banco de registros que se utilizará para guardar el dato o para obtenerlo. La dirección de la memoria de datos que se utilizará se especifica en los bits intermedios (*Adr*) y, como está formada por 4 bits, la memoria de datos será de $2^4$ = 16 palabras. La memoria de datos es una memoria RAM tal y como se ha presentado en el módulo anterior, con una entrada para las direcciones (*M@*) de 4 bits y otra para indicar cuál es la operación que se debe realizar (*L'*/*E*) y con una entrada/salida de datos (*Md*). Se supone que la memoria realiza la operación en un ciclo de reloj.

Por lo tanto, es necesario que ampliéis el diagrama de flujo de la figura 48 para representar el funcionamiento de una máquina algorítmica capaz de procesar programas descritos con el repertorio de instrucciones anterior. Hay que recordar que, en esta figura, *Q* es la entrada de la memoria de programa que contiene la instrucción que se debe ejecutar.

**8.** Indicad qué modificaciones serían necesarias para adaptar la microarquitectura del YASP (figura 54) a un repertorio de instrucciones capaz de trabajar con direcciones de 16 bits y, así, poder disponer de una memoria de hasta 64 kB. El formato de las instrucciones que tienen un campo de direcciones ocuparía, en este caso, 3 bytes: uno para el código de operación y dos para las direcciones. Ello incluye la de salto incondicional y excluye las instrucciones con modo de direccionamiento inmediato. ¿Qué se debería tocar en el camino de datos? ¿Qué implicaciones tendría en la unidad de control?

# Solucionario

## Actividades

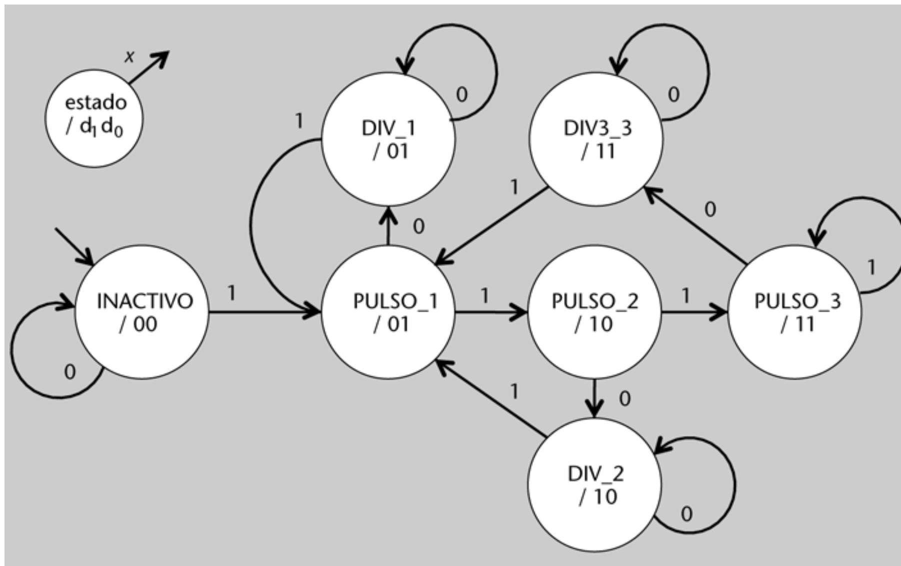
**1.** Para realizar el grafo de estados, hay que empezar por el estado inicial (INACTIVO) y decidir los estados siguientes, según todas las posibilidades de combinaciones de entradas. En este caso, solo pueden ser $x = 0$ o $x = 1$. Mientras no se detecte ningún cambio en la entrada ($x = 0$), la máquina permanece inactiva. En el instante en el que llegue un 1, se debe pasar a un nuevo estado para recordar que se ha iniciado un pulso (PULSO_1). A partir de ese momento, se llevarán a cabo transiciones hacia otros estados (PULSO_2 y PULSO_3), mientras $x = 1$. De esta manera, la máquina descubre si la amplitud del pulso es de uno, dos, tres o más ciclos de reloj.

En el último de estos estados (PULSO_3) se debe mantener hasta que no acabe el pulso de entrada, tenga la amplitud que tenga. En otras palabras, una vez $x$ haya sido 1 durante tres ciclos de reloj, ya no se distinguirá si la amplitud del pulso es de tres o más ciclos de reloj.

La máquina debe recordar la amplitud del último pulso recibido, de manera que la salida $D$ se mantenga en 01, 10 o 11, según el caso. Para ello, en los estados de detección de amplitud de pulso (PULSO_1, PULSO_2 y PULSO_3), cuando se recibe un cero de finalización de pulso ($x = 0$) se pasa a un estado de mantenimiento de la salida que recuerda cuál es el factor de división de la frecuencia del reloj: DIV_1, DIV_2 y DIV_3.

Desde cualquiera de estos estados se debe pasar a PULSO_1 cuando se recibe, de nuevo, otro 1 en la entrada.

Figura 67. Grafo de estados del detector de velocidad de transmisión




Del grafo de estados de la figura 67 se puede deducir la tabla de transiciones siguiente:

| Estado actual | | | | Entrada | Estado siguiente | | | |
|---|---|---|---|---|---|---|---|---|
| Identificación | $q_2$ | $q_1$ | $q_0$ | $x$ | Identificación | $q_2^+$ | $q_1^+$ | $q_0^+$ |
| INACTIVO | 0 | 0 | 0 | 0 | INACTIVO | 0 | 0 | 0 |
| INACTIVO | 0 | 0 | 0 | 1 | PULSO_1 | 0 | 0 | 1 |
| PULSO_1 | 0 | 0 | 1 | 0 | DIV_1 | 1 | 0 | 1 |
| PULSO_1 | 0 | 0 | 1 | 1 | PULSO_2 | 0 | 1 | 0 |
| PULSO_2 | 0 | 1 | 0 | 0 | DIV_2 | 1 | 1 | 0 |
| PULSO_2 | 0 | 1 | 0 | 1 | PULSO_3 | 0 | 1 | 1 |
| PULSO_3 | 0 | 1 | 1 | 0 | DIV_3 | 1 | 1 | 1 |
| PULSO_3 | 0 | 1 | 1 | 1 | PULSO_3 | 0 | 1 | 1 |
| DIV_1 | 1 | 0 | 1 | 0 | DIV_1 | 1 | 0 | 1 |
| DIV_1 | 1 | 0 | 1 | 1 | PULSO_1 | 0 | 0 | 1 |

| Estado actual | | | | Entrada | Estado siguiente | | | |
|---|---|---|---|---|---|---|---|---|
| **Identificación** | $q_2$ | $q_1$ | $q_0$ | $x$ | **Identificación** | $q_2^+$ | $q_1^+$ | $q_0^+$ |
| DIV_2 | 1 | 1 | 0 | 0 | DIV_2 | 1 | 1 | 0 |
| DIV_2 | 1 | 1 | 0 | 1 | PULSO_1 | 0 | 0 | 1 |
| DIV_3 | 1 | 1 | 1 | 0 | DIV_3 | 1 | 1 | 1 |
| DIV_3 | 1 | 1 | 1 | 1 | PULSO_1 | 0 | 0 | 1 |

En este caso, es conveniente realizar una codificación de los estados que respete el hecho de que el estado inicial sea el 000, pero que la salida se pueda obtener directamente de la codificación del estado. (En el ejercicio, esto no se pide y, por lo tanto, solo se debe tomar como ejemplo.)

La tabla de verdad para las salidas es la siguiente:

| Estado actual | | | | Salida |
|---|---|---|---|---|
| **Identificación** | $q_2$ | $q_1$ | $q_0$ | $D$ |
| INACTIVO | 0 | 0 | 0 | 00 |
| PULSO_1 | 0 | 0 | 1 | 01 |
| PULSO_2 | 0 | 1 | 0 | 10 |
| PULSO_3 | 0 | 1 | 1 | 11 |
| DIV_1 | 1 | 0 | 1 | 01 |
| DIV_2 | 1 | 1 | 0 | 10 |
| DIV_3 | 1 | 1 | 1 | 11 |

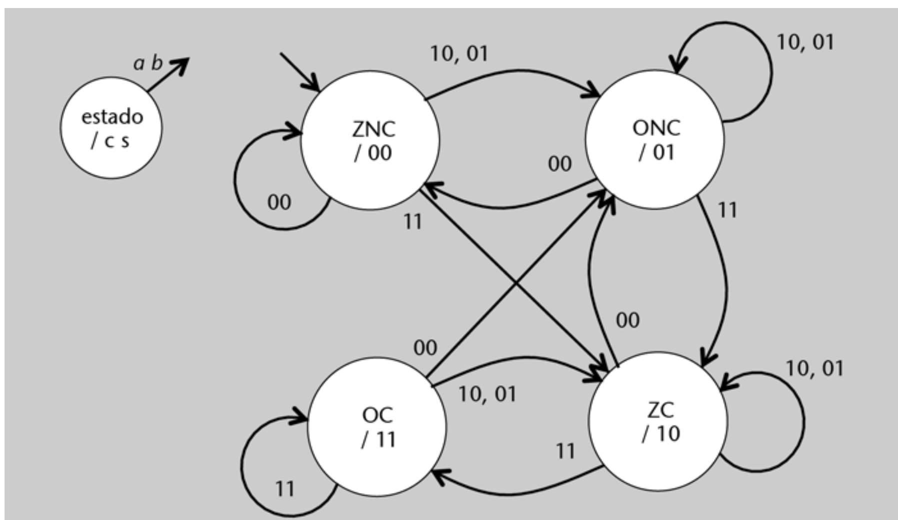
**2.** El estado de un sumador, en un momento determinado, lo fija tanto el resultado de la suma como del acarreo de salida del periodo de reloj anterior, que es el que hay que recordar. Por lo tanto, la máquina de estados correspondientes empieza en un estado en el que se supone que la suma previa ha sido $(c, s) = 0 + 0$, donde '+' indica la suma aritmética de los dos bits.

Se queda en el estado inicial (ZNC, de *zero and no carry*) mientras los bits que se han de sumar sean (0, 0). Si uno de los dos está en 1 y el otro no, entonces la suma da 1 y el bit de acarreo, 0; por tanto, hay que pasar al estado correspondiente (ONC, de *one and no carry*). El caso que falta es (1, 1), que genera acarreo para la suma siguiente, pero el resultado de la suma es 0. En consecuencia, de ZNC pasa a ZC (*zero and carry*).

Una reflexión similar se puede hacer estando en ONC, en OC (*one and carry*) y en ZC. Hay que tener presente que, una vez hecha, se debe comprobar que se han previsto todos los casos de entrada.

En la figura 68 aparece el grafo de estados correspondiente. Las entradas $(a, b)$ se denotan de manera compacta con los dos bits consecutivos.

Figura 68. Grafo de estados del sumador en serie



Del grafo de estados de la figura 68 se puede deducir la tabla de transiciones siguiente:

| Estado actual | | | Entrada | | Estado siguiente | | |
|---|---|---|---|---|---|---|---|
| **Identificación** | $q_1$ | $q_0$ | $a$ | $b$ | **Identificación** | $q_1^+$ | $q_0^+$ |
| ZNC | 0 | 0 | 0 | 0 | ZNC | 0 | 0 |
| ZNC | 0 | 0 | 0 | 1 | ONC | 0 | 1 |
| ZNC | 0 | 0 | 1 | 0 | ONC | 0 | 1 |
| ZNC | 0 | 0 | 1 | 1 | ZC | 1 | 0 |
| ONC | 0 | 1 | 0 | 0 | ZNC | 0 | 0 |
| ONC | 0 | 1 | 0 | 1 | ONC | 0 | 1 |
| ONC | 0 | 1 | 1 | 0 | ONC | 0 | 1 |
| ONC | 0 | 1 | 1 | 1 | ZC | 1 | 0 |
| ZC | 1 | 0 | 0 | 0 | ONC | 0 | 1 |
| ZC | 1 | 0 | 0 | 1 | ZC | 1 | 0 |
| ZC | 1 | 0 | 1 | 0 | ZC | 1 | 0 |
| ZC | 1 | 0 | 1 | 1 | OC | 1 | 1 |
| OC | 1 | 1 | 0 | 0 | ONC | 0 | 1 |
| OC | 1 | 1 | 0 | 1 | ZC | 1 | 0 |
| OC | 1 | 1 | 1 | 0 | ZC | 1 | 0 |
| OC | 1 | 1 | 1 | 1 | OC | 1 | 1 |

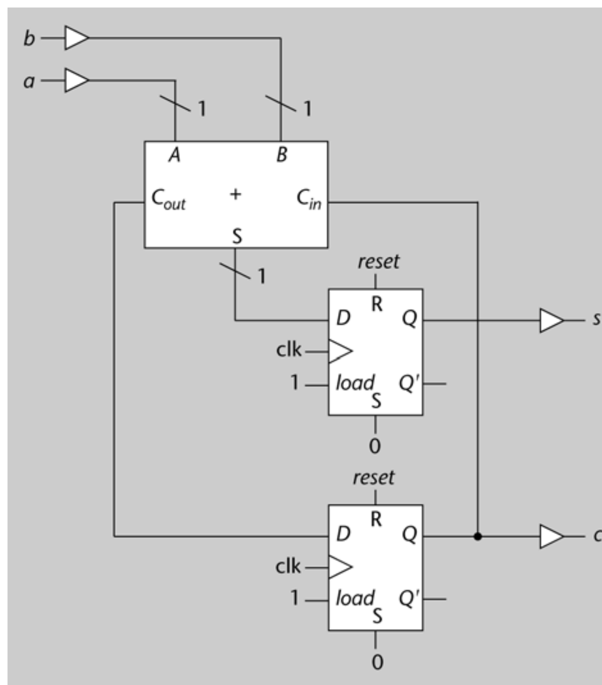
En este caso, la codificación de los estados se ha hecho respetando las salidas del sumador en serie. Por lo tanto, $c = q_1$ y $s = q_0$, como se puede comprobar en la tabla anterior.

Las funciones de excitación son, de hecho, coincidentes con las de un sumador completo:

$$q^+{}_1 = q_1\ (a \oplus b) + ab$$
$$q^+{}_0 = (a \oplus b) \oplus q_1$$

La implementación, que no se pide en el ejercicio, se da a modo de ejemplo de construcción de una máquina de estados con bloques combinacionales.

Figura 69. Circuito secuencial de suma de números en serie



**3.** En primer lugar, hay que elaborar las tablas de transición y de salidas. En este caso, la de salidas se puede deducir de la codificación que se explica a continuación.

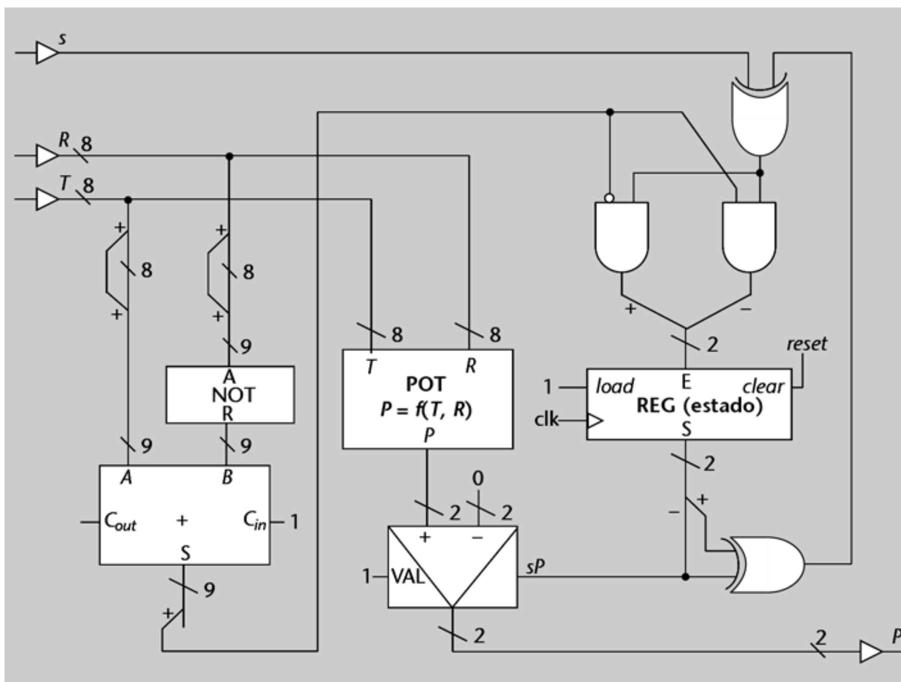
La codificación de los estados se realiza según el número binario correspondiente, teniendo en cuenta que el cero se reserva para el estado inicial (APAGADO). La codificación de la entrada de control será un bit para la señal de conmutación *s*. Y la condición ($T - R < 0$) se expresará con una señal que se denominará *c*, que será 1 cuando se cumpla. Finalmente, será necesario un selector de valor para *P*, que será *sP*. Cuando *sP* sea 0, $P = 0$ y cuando sea 1, $P = f(T, R)$.

Del grafo de estados de la figura 9 se puede deducir la tabla de transiciones siguiente:

| Estado actual | | | Entradas | | Estado siguiente | | |
|---|---|---|---|---|---|---|---|
| Identificación | $q_1$ | $q_0$ | $s$ | $c$ | Identificación | $q_1^+$ | $q_0^+$ |
| APAGADO | 0 | 0 | 0 | x | APAGADO | 0 | 0 |
| APAGADO | 0 | 0 | 1 | 0 | ENCENDIDO | 1 | 0 |
| APAGADO | 0 | 0 | 1 | 1 | CALENTANDO | 0 | 1 |
| CALENTANDO | 0 | 1 | 1 | x | APAGADO | 0 | 0 |
| CALENTANDO | 0 | 1 | 0 | 0 | ENCENDIDO | 1 | 0 |
| CALENTANDO | 0 | 1 | 0 | 1 | CALENTANDO | 0 | 1 |
| ENCENDIDO | 1 | 0 | 1 | x | APAGADO | 0 | 0 |
| ENCENDIDO | 1 | 0 | 0 | 0 | ENCENDIDO | 1 | 0 |
| ENCENDIDO | 1 | 0 | 0 | 1 | CALENTANDO | 0 | 1 |

A partir de esta tabla se pueden obtener las funciones $q_1^+$ y $q_0^+$. La salida *sP* coincide con $q_0$, ya que solo hay que poner una potencia del radiador diferente de cero en el estado de CALENTANDO, que es el único que tiene $q_0$ a 1. El esquema que aparece a continuación es el del circuito secuencial correspondiente.

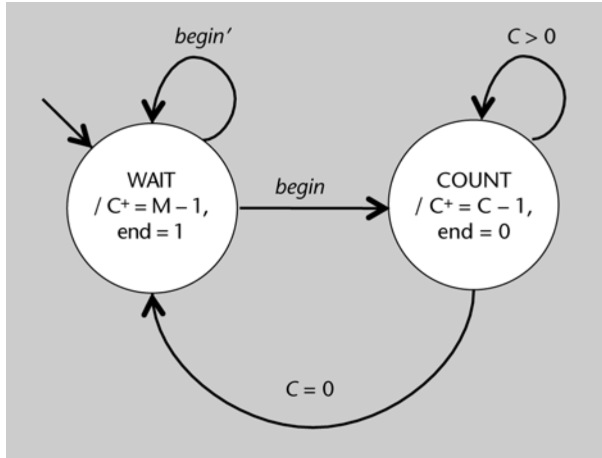
Figura 70. Controlador de un calefactor con termostato




El cálculo de *c* se realiza con el bit de signo del resultado de la operación $T - R$. Para garantizar que no haya problemas de desbordamiento (caso de una temperatura negativa, por ejemplo), se amplía el número de bits a 1 extendiendo el bit de signo.

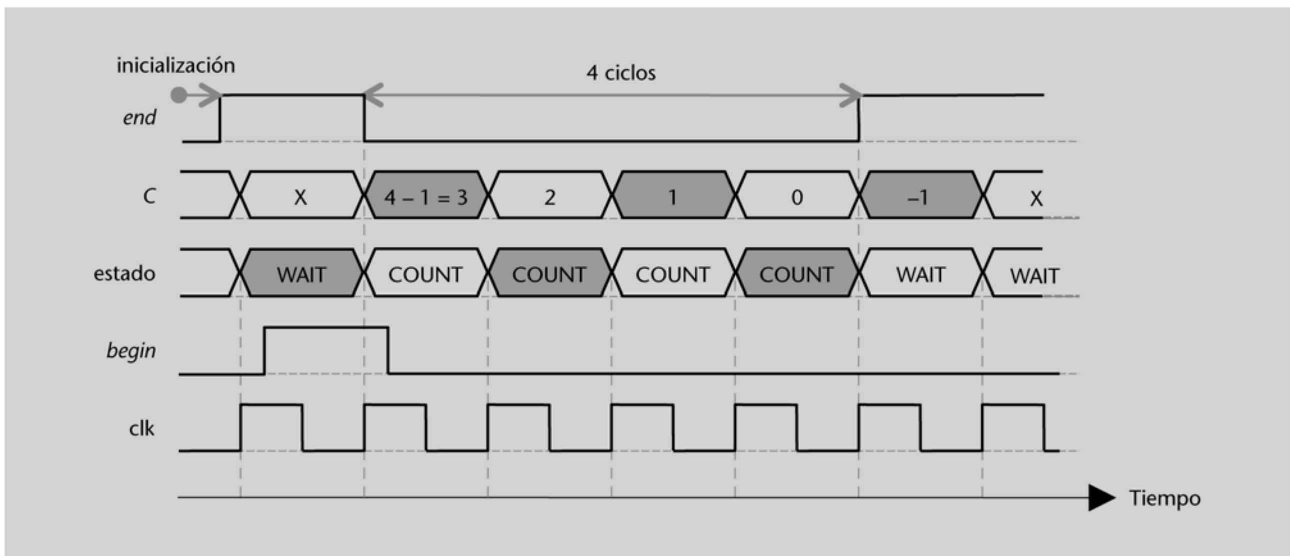
**4.** En este caso, en el estado de espera WAIT se deben ir efectuando cargas de valores decrementados ($M - 1$) en el registro de cuenta $C$ hasta que *begin* = 1. Entonces se debe pasar al estado de ir contando, COUNT. En este estado, decrece $C$ y, si el último valor de $C$ ha sido cero, se acaba la cuenta. El diagrama de transiciones de estados es, pues, muy similar al del contador de la figura 10:

Figura 71. Grafo de estados de un contador hacia atrás "programable"




Hay que tener presente que, tal como sucede con el contador incremental, el registro $C$ acaba cargando un valor de más que la cuenta que hace. En este caso, pasa a tener el valor –1 después de volver al estado WAIT. Este valor se pierde en el ciclo siguiente, con la carga de un posible nuevo valor $M$.

Figura 72. Cronograma de ejemplo para el contador atrás




El circuito correspondiente a este contador es más sencillo que el del contador incremental, ya que basta con un único sumador que realice las funciones de restador para decrecer el próximo valor de $C$ y un único registro. Dado que el grafo de estados es equivalente, la función que calcula el estado siguiente es la misma. En este caso, la condición ($C < B$) se transforma en ($C > 0$).

Figura 73. Circuito secuencial correspondiente al contador atrás

**5.** La EFSM del velocímetro tiene un punto en común con el contador: la velocidad es, de hecho, el número de veces que $c = 1$ hasta que $s = 1$. En este caso, sin embargo, no se debe contar en cada ciclo y, por lo tanto, se necesitarán dos estados diferentes para tener en cuenta si $c = 1$ (COUNT) o $c = 0$ (WAIT). Además, la condición de parada de la cuenta no es una condición sobre una variable, sino sobre una entrada (*s*).

El hecho de que el valor de la velocidad se deba mantener durante todo un segundo implica que es necesaria una variable, *V*, para almacenarla. Esta variable se actualizará cada vez que pase un segundo, es decir, cada vez que $s = 1$.

Para su cálculo será necesaria una variable que almacene el número de centímetros que se han detectado, *C*.

Si se produce el caso de que, en un mismo periodo de reloj, *s* y *c* están en 1, ello se deberá tener en cuenta: el centímetro se puede tener en cuenta o bien en el segundo actual o bien en el posterior.

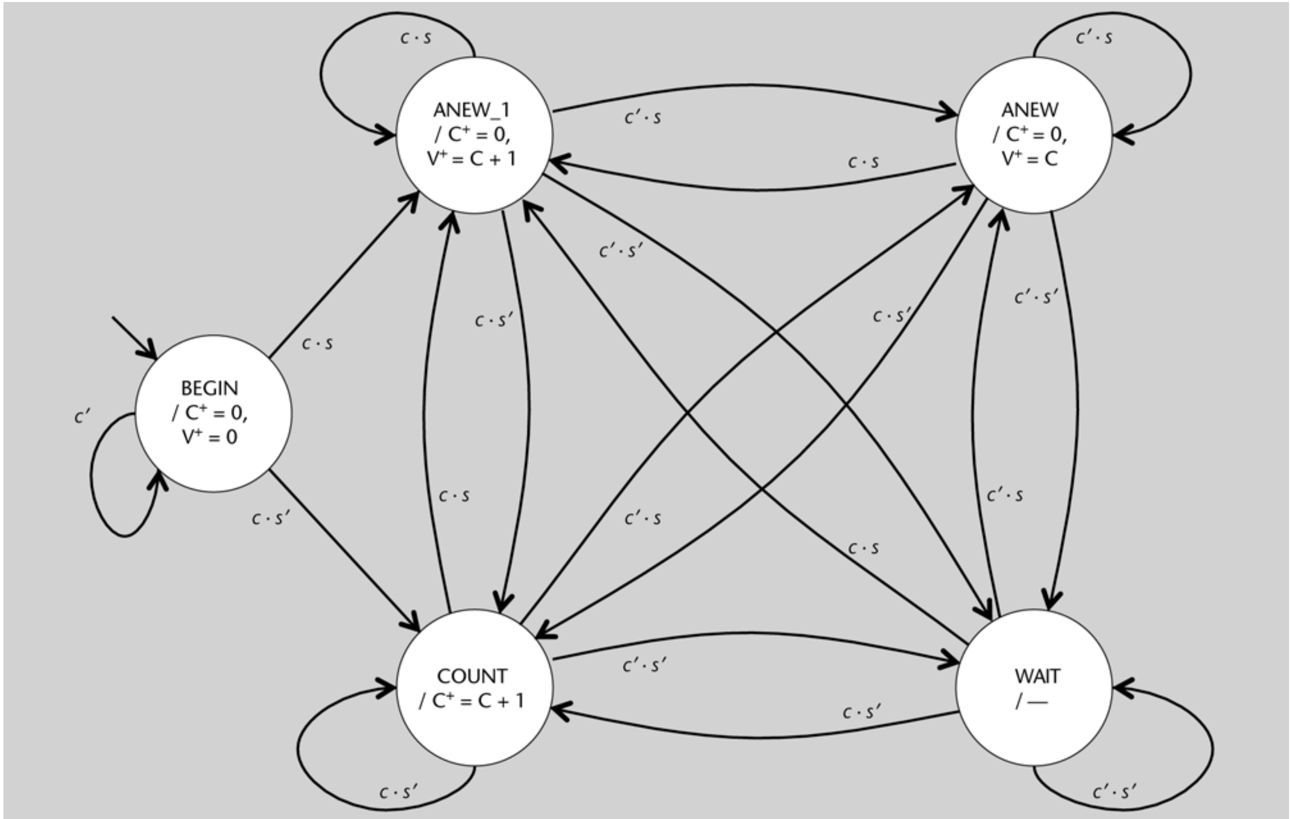
De hecho, cuando $s = 1$, habrá dos estados que reiniciarán la cuenta: ANEW, si no hay simultaneidad con *c*, y ANEW_1, si la hay. En el último caso se puede optar por inicializar el contador *C* a 1, que significa que el centímetro se contaría en el segundo posterior, o por acumularlo en la cuenta del segundo actual, que es lo que se hace en el grafo que se muestra a continuación.

Aunque tiene una apariencia complicada, las transiciones son sencillas, ya que hay que pensar en las acciones asociadas a cada estado. Así, a COUNT se va cada vez que hay que contar un centímetro sin que haya pasado un segundo (*cs'*); a WAIT, cada vez que las entradas están en cero (*c's'*); a ANEW, cada vez que se acaba una cuenta y se debe actualizar *V* (*sc'*) sin considerar un centímetro simultáneo, cosa que se tiene en cuenta en ANEW_1.

Cabe señalar que el grafo de estados se puede simplificar si se supone que, desde el principio, se ejecuta un *reset* en el circuito. En ese caso, los registros *C* y *V* también se pondrían a cero y ya no sería necesario el estado BEGIN. El estado inicial debería ser, en este caso, el de espera, WAIT.

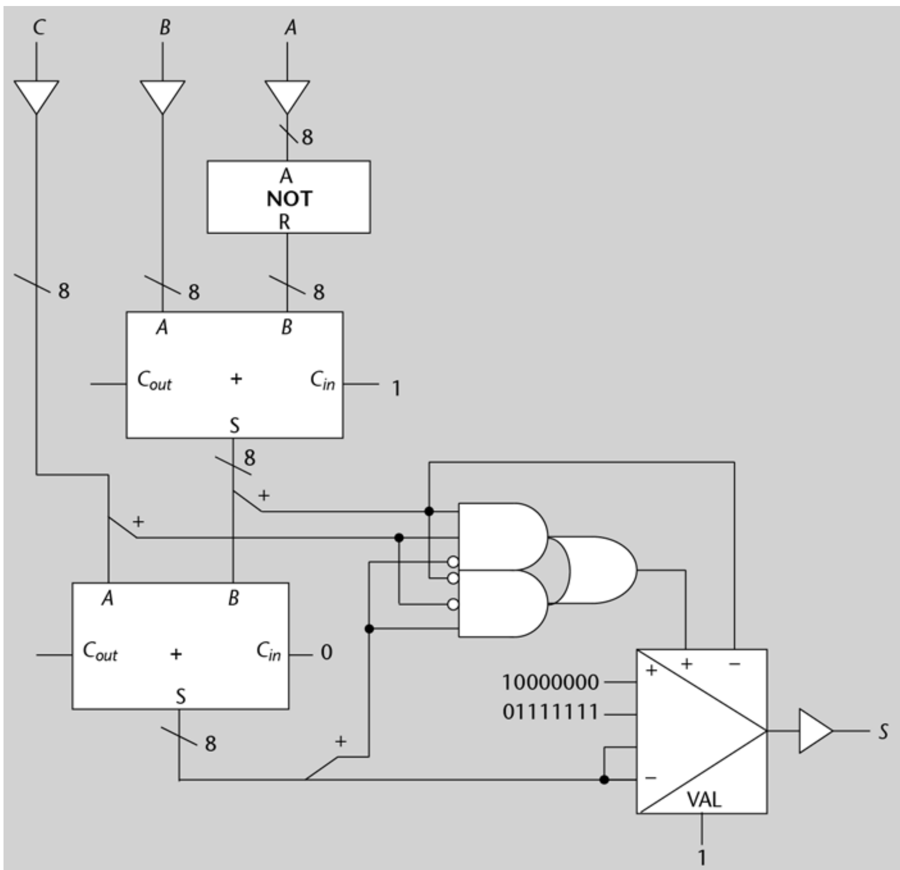
A modo de curiosidad, se podría utilizar el contador de la actividad anterior de manera que contara tantos ciclos de reloj como caben en un segundo. De este modo, el circuito conjunto solo necesitaría una entrada que le indicara cuándo se ha avanzado un centímetro, además de la de *reset* y de reloj, por supuesto.

Figura 74. Grafo de estados del velocímetro



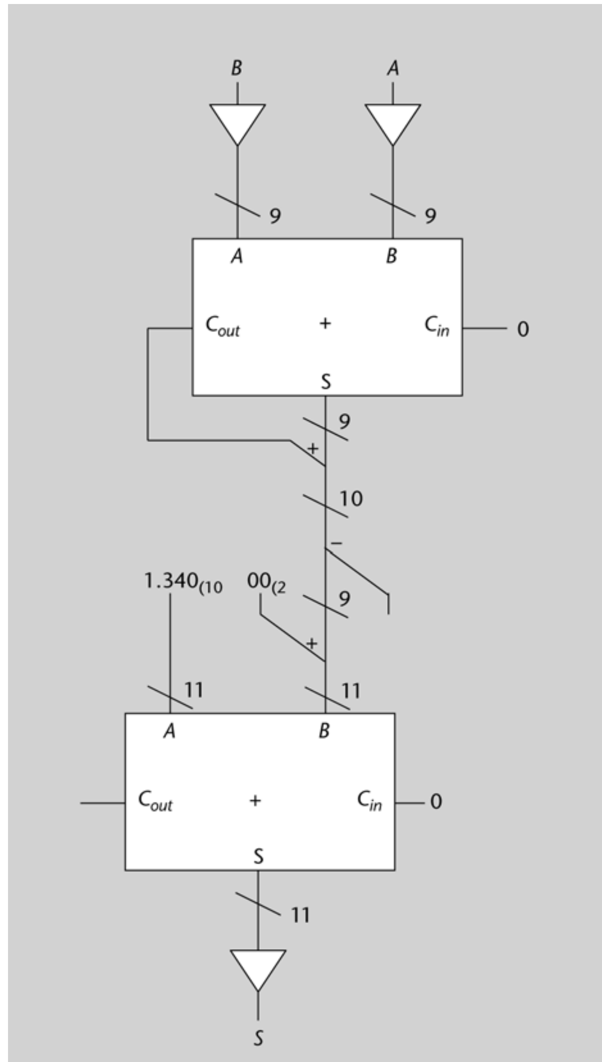

**6.** El módulo que calcula $C + B - A$ puede incurrir en desbordamiento. Para simplificar el diseño, primero se hace la resta, que nunca lo genera, y después la suma. En el caso de que se produzca desbordamiento, el resultado será el número mayor o el más pequeño posible, según sea la suma de números positivos o negativos, respectivamente.

Figura 75. Módulo de suma de error en la referencia del controlador de velocidad



El módulo que calcula 1.340 + ($A$ + $B$)/2 no tiene problemas de desbordamiento porque el resultado puede ser, como máximo, 1.340 + ($2^9$ – 1) = 1.851, que es más pequeño que el número mayor que se puede representar con 11 bits, $2^{11}$ – 1 = 2.047. De hecho, el valor máximo del resultado es 1.340 + 320 = 1.660, considerando la función de la figura 20. La división por 2 se realiza desplazando el resultado de la suma de $A$ y $B$ a la derecha, descartando el bit menos significativo, pero se debe tener en cuenta que la suma puede generar un bit de acarreo que se ha de incluir en el valor pendiente de desplazar. Para ello, primero se forma un número de 10 bits con el bit de acarreo y, después, se hace el desplazamiento a la derecha.

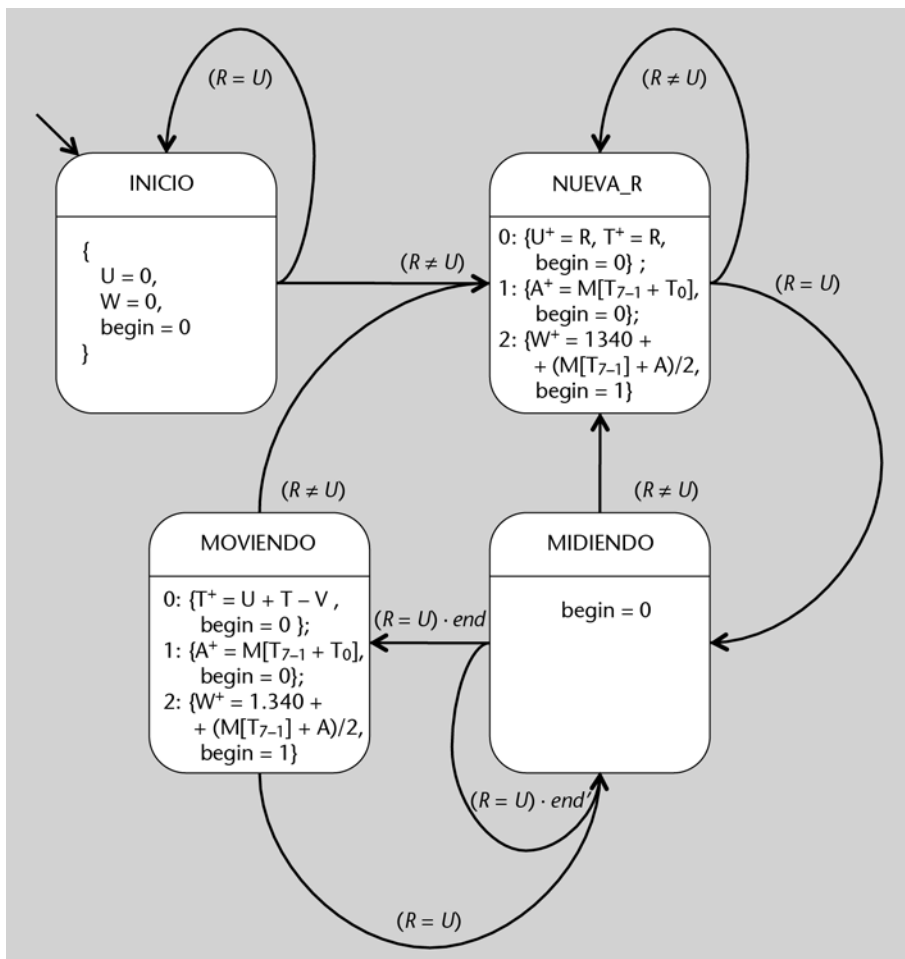
Figura 76. Módulo de cálculo de la amplitud de pulso del controlador de velocidad




**7.** La detección de si hay que cargar una nueva referencia o no se puede hacer comparando el valor actual de la variable $U$ con el de la entrada $R$. Si es igual, significa que no hay cambios. Si es diferente, se debe hacer una carga de un nuevo valor. Por lo tanto, $R \neq U$ es equivalente a *ld_r*. Así, la modificación en el modelo del PSM consiste en sustituir *ld_r* por ($R \neq U$) y *ld_r*' por ($R = U$).

El circuito correspondiente sería muy similar, pero debería incorporar un comparador entre $U$ y $R$ para generar, de manera interna, la señal *ld_r*.

Figura 77. Modelo modificado de la PSM del control adaptativo de velocidad




**8.** La tabla de verdad se puede construir a partir de la representación de la ASM correspondiente en la figura 31.

| **Estado actual** | | **Entradas** | | | | | | **Estado siguiente** | |
|---|---|---|---|---|---|---|---|---|---|
| **Identificación** | $S_{2-0}$ | *d* | *t* | *r* | *s* | *e* | *h* | **Identificación** | $S^{+}_{2-0}$ |
| IDLE | 000 | 0 | x | x | x | x | x | IDLE | 000 |
| IDLE | 000 | 1 | 0 | x | x | x | x | UP | 100 |
| IDLE | 000 | 1 | x | 0 | x | x | x | UP | 100 |
| IDLE | 000 | 1 | 1 | 1 | x | x | x | MORE | 001 |
| MORE | 001 | x | x | x | x | x | x | HEAT | 101 |
| HEAT | 101 | x | x | x | 0 | 0 | x | HEAT | 101 |
| HEAT | 101 | x | x | 0 | 0 | 1 | 0 | UP | 100 |
| HEAT | 101 | x | x | x | 1 | x | x | UP | 100 |
| HEAT | 101 | x | x | 0 | 0 | 1 | 1 | KEEP | 110 |
| HEAT | 101 | x | x | 1 | 0 | 1 | x | MORE | 001 |
| KEEP | 110 | x | x | x | 0 | x | x | KEEP | 110 |
| KEEP | 110 | x | x | x | 1 | x | x | UP | 100 |
| UP | 100 | x | x | x | x | x | x | IDLE | 000 |

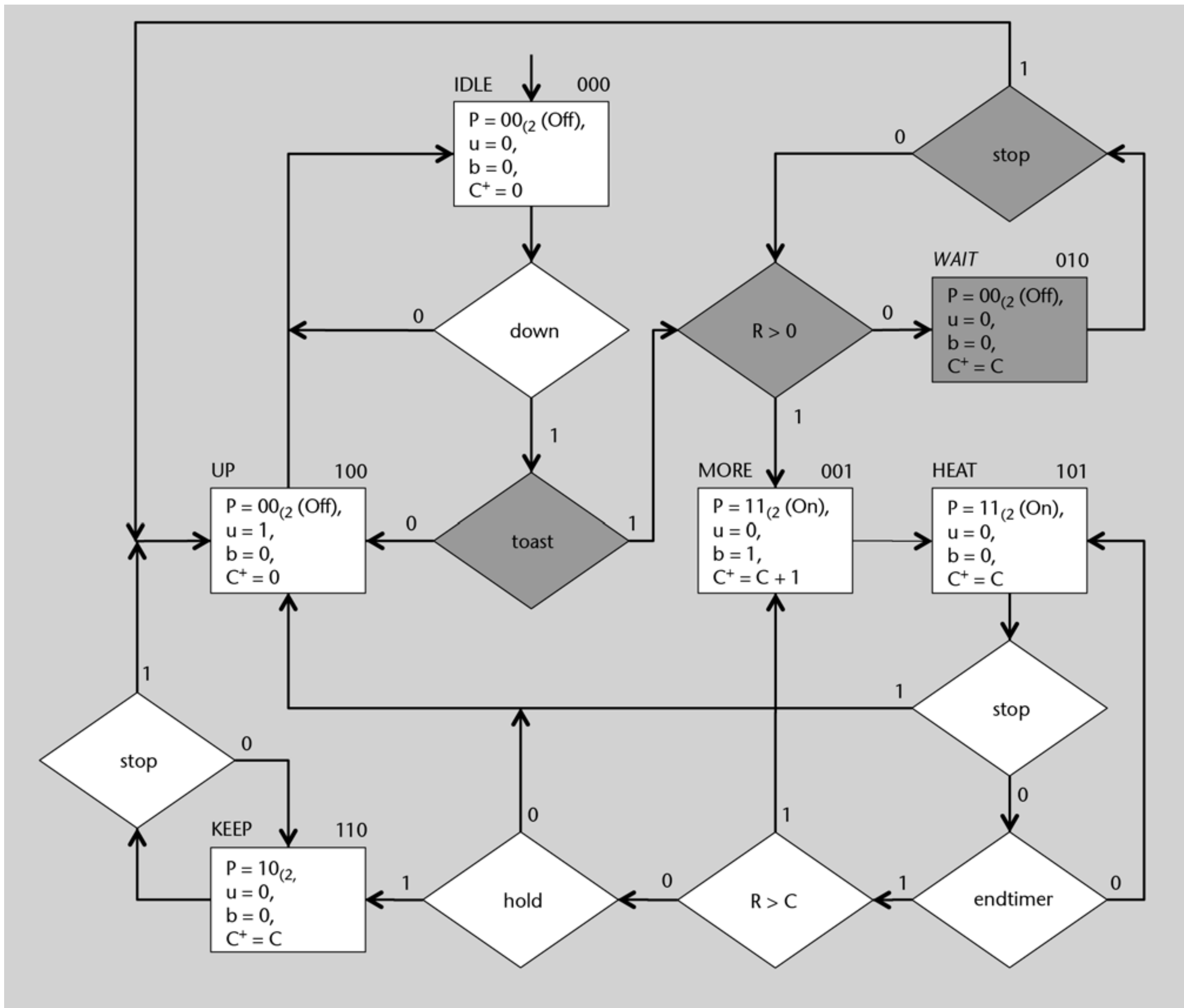
Se puede comprobar que existen muchos *don't-cares* que pueden ayudar a obtener expresiones mínimas para las funciones de cálculo del estado siguiente, pero también que son difíciles de manipular de manera manual.

**9.** La modificación implica dividir en dos la caja de decisión de *toast* & ($R > 0$), de modo que primero se pregunte sobre si se ha detectado tostada y, en caso afirmativo, se compruebe si $R > 0$, para pasar a MORE. En el caso de que $R$ sea 0, se pasará a una nueva caja de estado (WAIT) en la que se esperará a que se apriete el botón de parada (*stop*) o se gire la rueda a una posición diferente de 0.

En este caso, es más complicado encontrar una codificación de estados de manera que, entre estados vecinos, exista el menor número de cambios posible. Especialmente porque el estado UP tiene 4 vecinos y, en una codificación de 3 bits, como mucho, hay tres vecinos posibles para cada código en el que solo se cambia un bit.

En la ASM que se ha cambiado, se ha aprovechado uno de los códigos libres para WAIT (010). Las acciones asociadas son las de mantener los elementos calefactores apagados sin hacer saltar la rebanada de pan ni activar el temporizador.

Figura 78. ASM del controlador de una tostadora con estado de espera



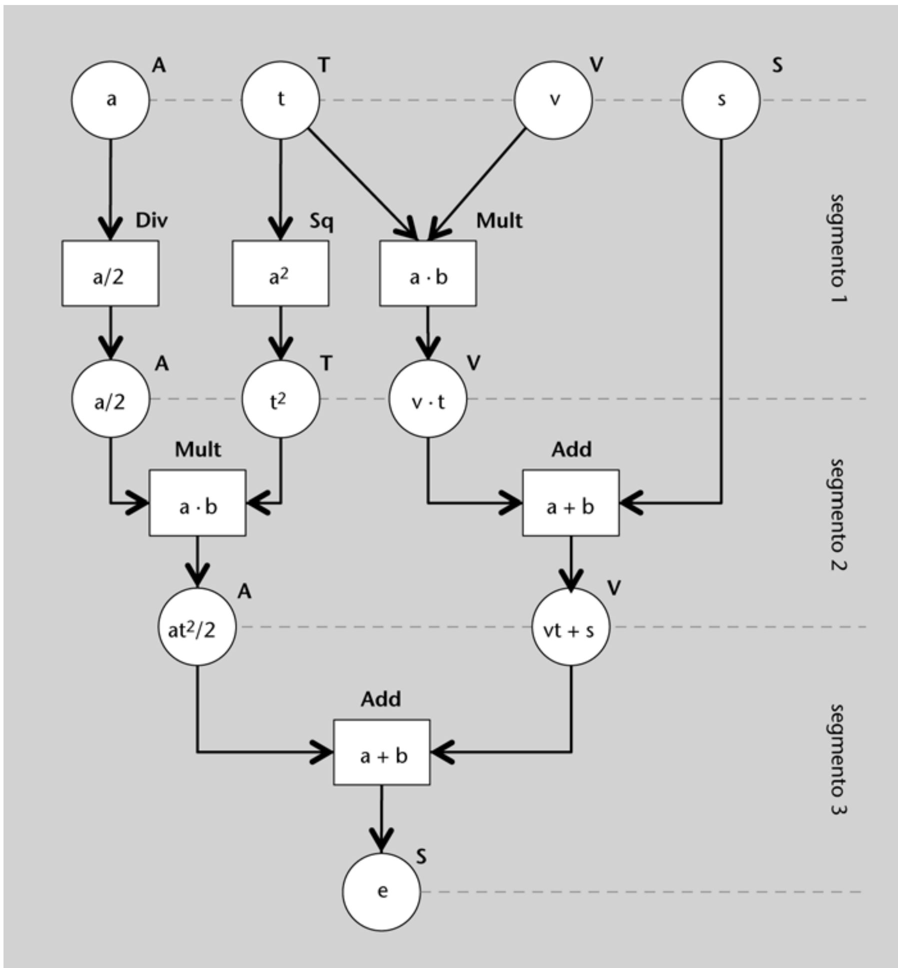

**10.** Antes de realizar el esquema de cálculo, es conveniente poner paréntesis a la expresión para agrupar las operaciones. Así, la expresión dada se puede transformar en:

$$(((a/2) \cdot (t^2)) + ((v \cdot t) + s))$$

A partir de la expresión anterior se puede generar fácilmente el diagrama del esquema de cálculo correspondiente. En este caso, se ha optado por segmentarlo hasta llegar a un buen compromiso entre el número de etapas (3) y el número de recursos de cálculo (4). Sin segmentación habrían hecho falta 6 operadores, pero el retraso sería, en la práctica, muy grande. Con segmentación máxima y aprovechando el multiplicador para hacer el cuadrado de $t$, sería necesaria una etapa más, hasta 4, y solo 3 recursos de cálculo. No obstante, el número de etapas también implica, finalmente, un tiempo de proceso elevado.

En la figura 79 se muestra la solución con un esquema de cálculo segmentado. El esquema también lleva etiquetas para las variables, asociándolas a registros. En este caso, se ha optado por un reaprovechamiento total. Para permitir un funcionamiento progresivo, sería necesario que en cada etapa se utilizaran registros diferentes. Así, con 10 registros sería posible iniciar un nuevo cálculo en cada ciclo de reloj.

Figura 79. Esquema de cálculo de $at^2/2 + vt + s$




**11.** En cuanto a los multiplexores vinculados a las entradas de los recursos de cálculo, hay que tener presente que se ocupan de seleccionar los datos que necesitan según el periodo de reloj en el que se esté. Así, las entradas en posición 0 de los multiplexores proporcionan a los recursos asociados los datos para el primer periodo de reloj, las entradas 1 corresponden a los datos del segundo periodo, y así hasta el máximo número de periodos de reloj que sea necesario. Ahora bien, no todos los recursos de cálculo se utilizan en todos los periodos de reloj y, por lo tanto, los resultados que generan en estos periodos de tiempo no importan, de la misma manera que tampoco importan qué entradas tengan. Así, se puede construir una tabla que ayude a aprovechar estos casos irrelevantes:

| Estado | << | + | CMP |
|---|---|---|---|
| *S*0 | *x* | *x* | *C* = *n* |
| *S*1 | *R* << 1 | *C* + 1 | *x* |
| *S*2 | *x* | *R* + *M* | *x* |
| *S*3 | *P* << 1 | *x* | *C* = *n* |

En el circuito de la figura 44 todos los *don't-cares* aparecían como 0 en las entradas de los multiplexores. Sin embargo, de cara a la optimización se deben considerar como tales. Así, se puede observar que el comparador puede estar conectado siempre a las mismas entradas y el multiplexor se puede suprimir. En el caso de la suma, bastaría con un multiplexor de dos entradas. Algo similar sucede con el decalador. Sin embargo, en este caso resulta más conveniente utilizar dos decaladores que tener un multiplexor. (Los decaladores son mucho más simples que los multiplexores, ya que no necesitan puertas lógicas.)

En el caso de los registros, se debe tener en cuenta que la elección entre mantener el valor y cargar otro nuevo se puede realizar con la señal de carga (*load*) correspondiente, lo que ayuda a reducir el orden de los multiplexores en su entrada. Es conveniente tener una tabla similar que ayude a visualizar las optimizaciones que se pueden llevar a cabo, como la que se muestra a continuación.

| Estado | *R* | *M* | *P* | *C* |
|---|---|---|---|---|
| *S*0 | 0 | *A* | *B* | 0 |
| *S*1 | *R* << 1 | *M* | *P* | *C* + 1 |
| *S*2 | *R* + *M* | *M* | *P* | *C* |
| *S*3 | *R* | *M* | *P* << 1 | *C* |

El caso más sencillo es el del registro *M*, que solo debe realizar una carga en el estado *S*0 y, por lo tanto, basta con utilizar *S*0 conectado a la señal correspondiente del registro asociado. Algo similar se puede hacer con *C*, si la puesta a cero se realiza aprovechando el *reset* del registro. En este caso, se haría un *reset* en el estado *S*0 y una carga en el estado *S*1. Con *P* es necesario utilizar un multiplexor de dos entradas para distinguir entre los dos valores que se pueden cargar (*B* o *P* << 1). Con *R* se puede aprovechar la misma solución si la puesta a cero se realiza con *reset*.

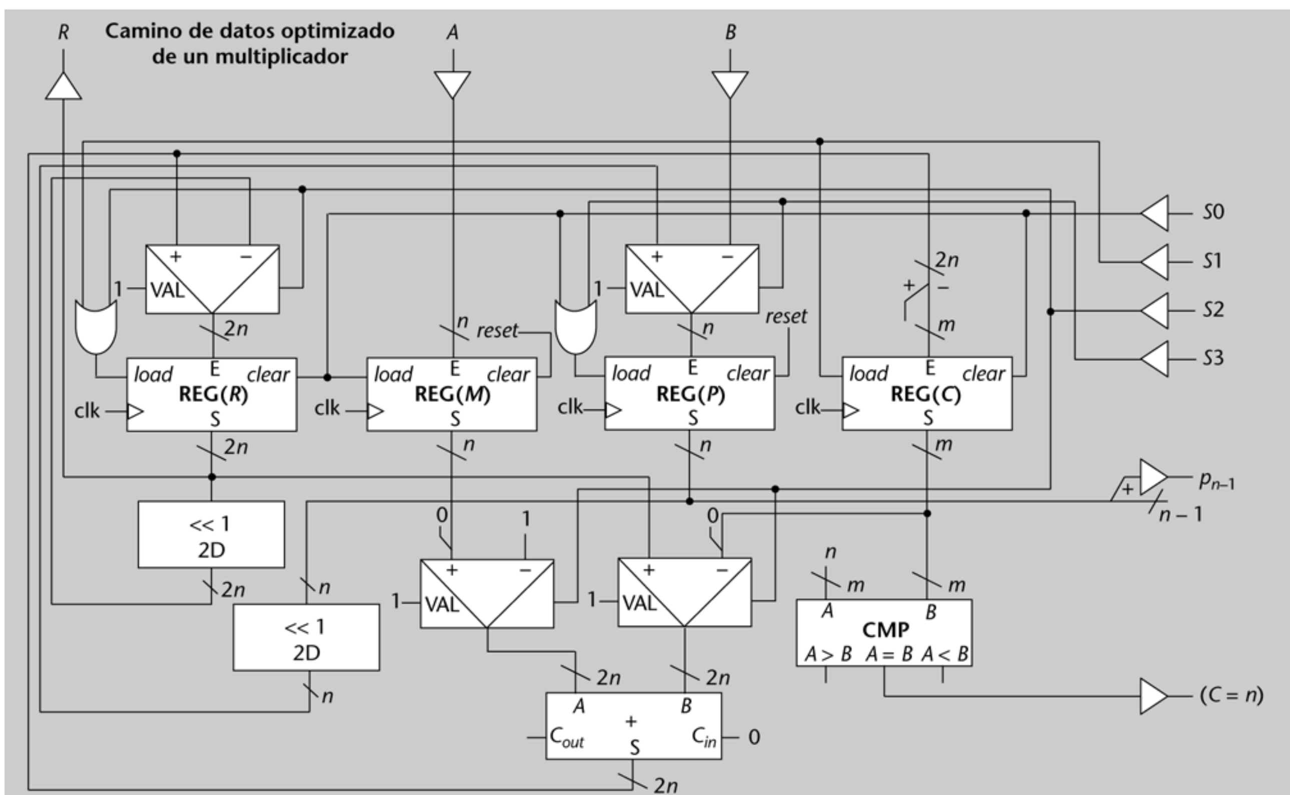
Finalmente, hay que tener en cuenta que las funciones lógicas que generan los valores de control de los multiplexores, de los *loads* y de los *reset* suelen ser más simples con la codificación *one-hot bit* que con la binaria. De hecho, en este caso no hay demasiada diferencia, pero utilizar la codificación *one-hot bit* ahorra el codificador de la unidad de control.

En la tabla siguiente se muestran las señales anteriores en función de los bits de estado. Aquellas operaciones en las que se carga un valor están separadas por puntos suspensivos, de manera que la activación de la señal de carga se muestra como "*P* =..." o "*R* =..." y la selección del valor que cargar como "...*B*", "...*P* << 1", "... *R* << 1" i "...*R* + *M*".

| Estado | + | *loadM* | *loadP* | *selP* | *loadC* | *resetC* | *loadR* | *selR* | *resetR* |
|---|---|---|---|---|---|---|---|---|---|
| *S*0 | (*C* + 1) | *M* = *A* | *P* = ... | ... *B* | — | *C* = 0 | — | ... *R* << 1 | *R* = 0 |
| *S*1 | *C* + 1 | (*M* = *M*) | (*P* = *P*) | ... *B* | *C* = *C* + 1 | — | *R* = ... | ... *R* << 1 | — |
| *S*2 | *R* + *M* | (*M* = *M*) | (*P* = *P*) | ... *B* | (*C* = *C*) | — | *R* = ... | ... *R* + *M* | — |
| *S*3 | (*C* + 1) | (*M* = *M*) | *P* = ... | ... *P* << 1 | (*C* = *C*) | — | (*R* = *R*) | ... *R* << 1 | — |

El circuito optimizado se muestra en la figura 80.

Figura 80. Unidad de procesamiento de un multiplicador en serie



**12.** Para completarla, se deben traducir los símbolos del programa a los códigos binarios correspondientes. De hecho, la codificación en binario de estos datos es bastante directa, siguiendo el formato de las instrucciones que interpreta el Femtoproc.

Para obtener la codificación en binario de una instrucción, se puede traducir a binario, de izquierda a derecha, primero el símbolo de la operación que hay que realizar (ADD, AND, NOT, JZ) y después los que representan los operandos, que pueden ser registros o direcciones. Si son registros, basta con obtener el número binario equivalente al número de registro que se especifica. Si son direcciones, se "desempaqueta" el número hexadecimal en el binario equivalente.

En la tabla siguiente se completa la codificación del programa del MCD.

| Dirección | Instrucción | Codificación | Comentario |
|---|---|---|---|
| ... | ... | ... | ... |
| 0Bh | `AND R0, R1` | 01 000 001 | Fuerza a que el resultado sea cero |
| 0Ch | `JZ 04h (sub)` | 11 000100 | Vuelve a hacer otra resta |
| pos,0Dh | `NOT R3, R4` | 10 011 100 | |
| 0Eh | `NOT R3, R3` | 10 011 011 | $R3 = R4 = R3 - R2$ |
| 0Fh | `JZ 12h (end)` | 11 010010 | |
| 10h | `AND R0, R1` | 01 000 001 | |
| 11h | `JZ 04h (sub)` | 11 000100 | Vuelve a hacer otra resta |
| end,12h | `ADD R5, R2` | 00 101 010 | |
| 13h | `ADD R5, R3` | 00 101 011 | $R5 = R2 + R3$, pero uno de los dos es cero |
| 14h | `AND R0, R1` | 01 000 001 | A la espera, hasta que se le haga *reset* |
| hlt,15h | `JZ 15h (hlt)` | 11 010101 | |

**13.** Con el repertorio de instrucciones del Femtoproc no es posible copiar el valor de un registro a otro diferente directamente. Pero se puede aprovechar la instrucción NOT para guardar en un registro el complemento del contenido de otro registro fuente. Después, basta con hacer una operación NOT del primer registro, de manera que este registro sea fuente y destino al mismo tiempo.

Esta situación se da en las instrucciones de las posiciones 0Dh y 0Eh del programa anterior:

| Dirección | Instrucción | Codificación | Comentario |
|---|---|---|---|
| ... | ... | ... | ... |
| 0Bh | `AND R0, R1` | 01 000 001 | Fuerza a que el resultado sea cero |
| 0Ch | `JZ 04h (sub)` | 11 000100 | Vuelve a hacer otra resta |
| pos,0Dh | `NOT R3, R4` | 10 011 100 | |
| 0Eh | `NOT R3, R3` | 10 011 011 | $R3 = R4 = R3 - R2$ |
| 0Fh | `JZ 12h (end)` | 11 010010 | |
| 10h | `AND R0, R1` | 01 000 001 | |
| ... | ... | ... | ... |

**14.** Hacer un desplazamiento a la izquierda de un bit de un número equivale a multiplicarlo por 2. La multiplicación por 2 se puede hacer con una suma. Así, si se quieren desplazar los bits del registro $R3$ un bit a la izquierda, basta con hacer `ADD R3, R3`.

**15.** Según el patrón de estos materiales y el ejemplo que se da, es necesaria una microorden de carga para cada registro y biestable: *ld_PC*, *ld_IR*, *ld_MB*, *ld_MAR*, *ld_A*, *ld_X*, *ld_C* y *ld_Z*. Como todos los registros solo tienen una entrada, no es necesario ningún selector para controlar un multiplexor adicional.

Ahora bien, los registros que cargan el dato proveniente del *BusW* cargan el resultado de llevar a cabo una operación en la ALU. Esta operación puede tener dos operandos, uno es el registro MB y otro puede ser PC, A o X. Por lo tanto, es necesaria una microorden de selección de cuál de estos tres registros transfiere su contenido al *BusR*. Esta microorden sería una señal de dos bits que se podría denominar *selBusR*.

En resumen, se necesitan 10 microórdenes; una de ellas (*selBusR*) es de 2 bits y otra, de selección de operación de la ALU, de 5. En total, 15 bits. Por lo tanto, sería compatible con el formato de microinstrucciones visto en el apartado 3.2.2 y, por ende, se podría implementar con secuenciador de la figura 53.

**16.** En este caso, se trataría de una instrucción de un único byte, ya que no hay ningún operando en memoria. Así, bastaría con leer la instrucción, comprobar que sea de incremento de X (supongamos que su símbolo es INX), realizar la operación y pasar a la instrucción siguiente. Esto último se puede hacer simultáneamente con la lectura del código de operación, como ya hemos visto.

| Etiqueta | Código de operación | μ–**instrucción** | Comentario |
|---|---|---|---|
| START: | EXEC | `MAR = PC` | **Fase 1**. Lectura de la instrucción |
| | EXEC | `MB = M[MAR], PC = PC + 1` | Carga del *buffer* de memoria e incremento del *PC* |
| | EXEC | `IR = MB` | |
| | JMPIF | `INX?, X_INX` | **Fase 2**. Descodificación: Si es INX, salta a X_INX |
| | | ... | |
| X_INX: | EXEC | `X = X + 1` | **Fase 3**. Ejecución de la operación |
| | JMPIF | `Inconditional, START` | Bucle infinito de interpretación |
| | | ... | |

**17.** El *pipe* se irá llenando hasta que se acabe la ejecución de la segunda instrucción. Una vez concluida la fase de ejecución de la operación (EO), el *pipe* se "vacía", de manera que las fases ya iniciadas de las instrucciones número 3, 4 y 5 no se continúan. Por tanto, el *pipeline* tendrá una latencia de 4 periodos de reloj más hasta ejecutar la operación de una nueva instrucción.

Figura 81. *Pipeline* de 4 etapas con salto de secuencia a la segunda instrucción

| Instrucción | Etapa del *pipeline* (fase del ciclo) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | LI | LA | CO | EO | | | | | |
| 2 | | LI | LA | CO | EO | | | | |
| 3 | | | LI | LA | CO | X | | | |
| 4 | | | | LI | LA | X | | | |
| 5 | | | | | LI | X | | | |
| 11 | | | | | | LI | LA | CO | EO |
| 12 | | | | | | | LI | LA | CO |
| Período | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Las "X" del *pipeline* indican que los valores de los registros en la salida de las etapas correspondientes no importan, ya que se deben calcular de nuevo.

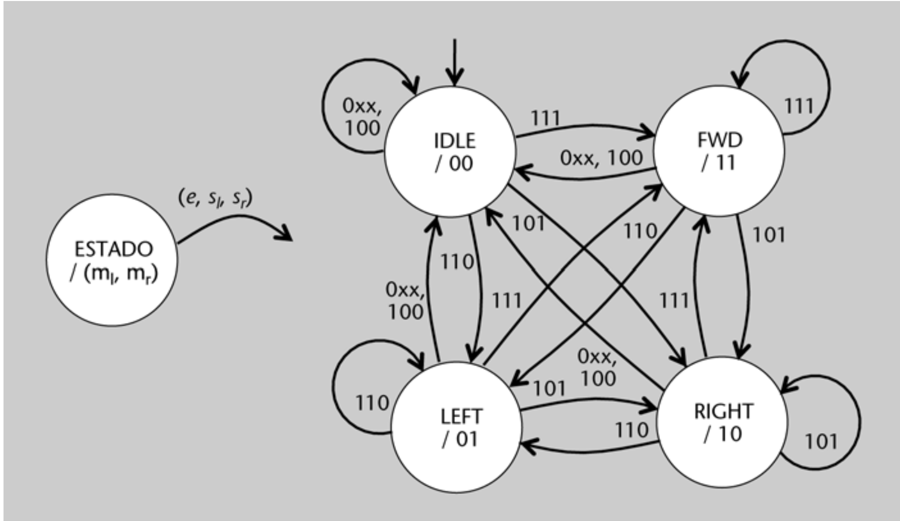
## Ejercicios de autoevaluación

**1.** Para realizar el grafo de estados que represente el comportamiento que se ha indicado en el enunciado, hay que empezar por el estado inicial, que se denominará IDLE porque, inicialmente, el robot no debe hacer nada. En este estado, la salida tiene que ser $(m_l, m_r)$ = (0, 0), ya que el vehículo debe permanecer inmóvil. Tanto si la entrada *e* es 0 como si es 1 y no se detecta línea en ninguno de los sensores que dan las señales de entrada, el robot se ha de mantener en este estado. De hecho, desde cualquier otro estado que tenga esta máquina de estados se debe pasar a IDLE cuando se detecte alguna de estas condiciones. Es decir, cuando $(e, s_l, s_r)$ sean (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1) o (1, 0, 0).

Desde el estado de IDLE, con $e = 1$, se pasa al estado de avanzar (FWD), si se detecta línea en los dos sensores, o a los de girar a la izquierda (LEFT) o a la derecha (RIGHT), si la línea solo se detecta en la entrada del sensor izquierdo o derecho, respectivamente. De manera similar, cada uno de estos tres estados debe tener un arco que vaya a los otros dos y uno reentrante para el caso correspondiente al mismo estado.

El dibujo del grafo correspondiente es un poco complicado porque todos los nodos están conectados entre sí.

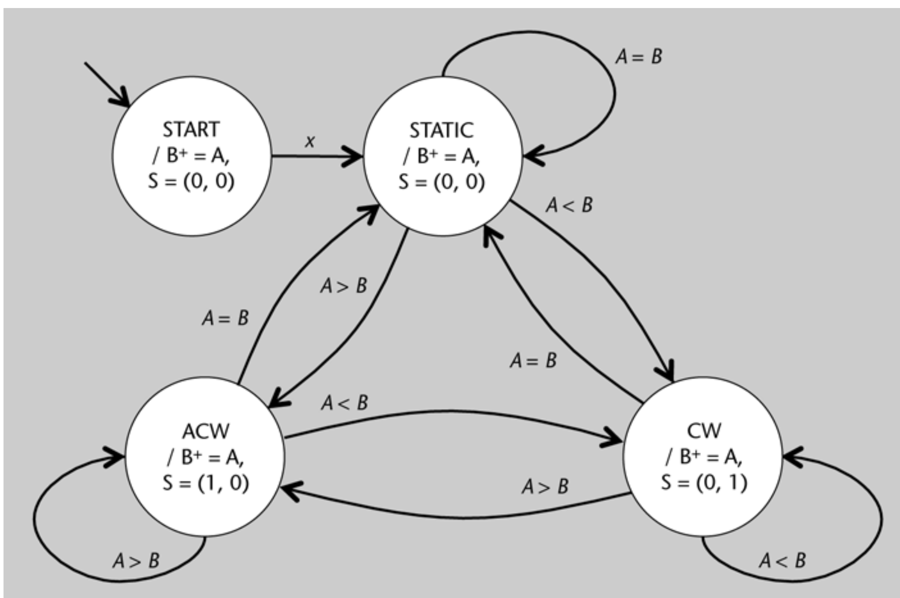
Figura 82. Grafo de estados del controlador de un seguidor de líneas




**2.** El enunciado del problema define tanto el comportamiento que debe seguir el detector como las entradas y salidas que tiene, así como su codificación. Ahora bien, se ha de tener en cuenta que las EFSM pueden utilizar variables. De hecho, como la detección del sentido de giro se realiza comparando la posición angular actual ($A$) con la anterior, la EFSM correspondiente debe tener una que la almacene, y que se denominará $B$. Así, si $A > B$, el eje gira en el sentido contrario a las agujas del reloj y la salida debe ser (1, 0).

Con esta información ya se puede establecer el grafo de estados, que empezará en el estado inicial START. En este estado, la salida debe ser (0, 0) y se ha de dar el valor inicial a $B$, que será el de la entrada $A$, es decir, $B^+ = A$. Como no es posible comparar el valor de la posición angular actual con la anterior, de este estado se debe pasar a otro estado en el que se pueda hacer una segunda lectura. Así, se puede pasar de START a STATIC con la misma salida. Desde STATIC se deberá pasar a ACW (del inglés *anticlockwise*, en el sentido contrario a las agujas del reloj), si $A > B$; a CW (del inglés *clockwise*, en el sentido de las agujas del reloj), si $A < B$, o permanecer en STATIC, si $A = B$. En todos los estados, hay que almacenar en $B$ la posición angular actual. El grafo correspondiente se muestra a continuación.

Figura 83. EFSM del detector de sentido de giro



Dado que la codificación binaria de las entradas y salidas ya queda definida en el enunciado del problema, solo queda codificar la variable *B* y los estados: *B* debe tener el mismo formato que *A*, es decir, ha de ser un número natural de 3 bits, y los estados se codifican según la numeración binaria, de manera que START = 00, ya que es el estado al que se debe pasar en caso de *reset*. Como se puede ver en la siguiente tabla de transiciones, se ha hecho que los estados ACW y CW tengan una codificación igual a la de las salidas *S* correspondientes.

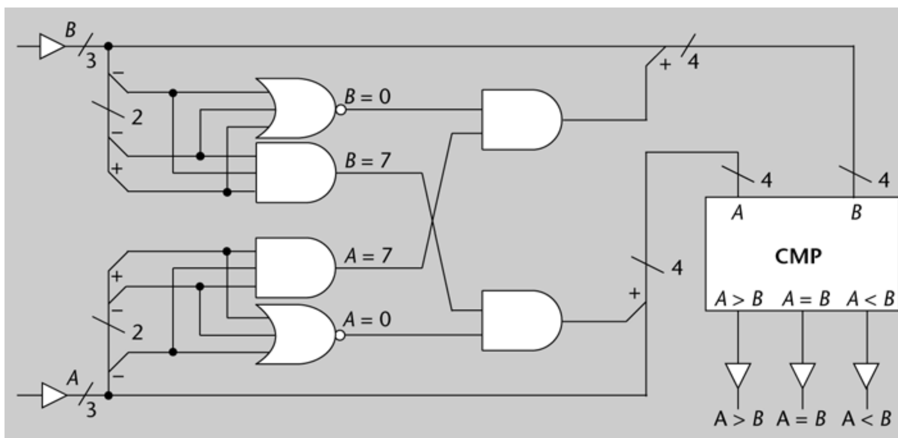
| Estado actual | | | Entradas | | | Estado siguiente | | |
|---|---|---|---|---|---|---|---|---|
| Identificación | $q_1$ | $q_0$ | $A > B$ | $A = B$ | $A < B$ | Identificación | $q_1^+$ | $q_0^+$ |
| START | 0 | 0 | x | x | x | STATIC | 1 | 1 |
| CW | 0 | 1 | 0 | 0 | 1 | CW | 0 | 1 |
| CW | 0 | 1 | 0 | 1 | 0 | STATIC | 1 | 1 |
| CW | 0 | 1 | 1 | 0 | 0 | ACW | 1 | 0 |
| ACW | 1 | 0 | 0 | 0 | 1 | CW | 0 | 1 |
| ACW | 1 | 0 | 0 | 1 | 0 | STATIC | 1 | 1 |
| ACW | 1 | 0 | 1 | 0 | 0 | ACW | 1 | 0 |
| STATIC | 1 | 1 | 0 | 0 | 1 | CW | 0 | 1 |
| STATIC | 1 | 1 | 0 | 1 | 0 | STATIC | 1 | 1 |
| STATIC | 1 | 1 | 1 | 0 | 0 | ACW | 1 | 0 |

En cuanto a las acciones relacionadas con cada estado, la asignación $B^+ = A$ se debe realizar en todos ellos y, consiguientemente, se hace de manera independiente al estado en el que esté. La función de salida *S* es muy sencilla:

$$S = (s_1, s_0) = (ACW, CW) = (q_1 \cdot q_0', q_1' \cdot q_0)$$

De cara a la implementación hay que tener en cuenta una cosa en cuanto al comparador: debe ser un “comparador circular”. Es decir, ha de tener en cuenta que del sector 7 se pasa, en el sentido contrario de las agujas del reloj, al 0 y que del 0 se pasa al 7. Por lo tanto, se debe cumplir que: {(0 < 1), (1 < 2), (2 < 3), ..., (5 < 6), (6 < 7), (7 < 0)}. Para que esto ocurra, una de las opciones es utilizar un comparador convencional de 4 bits en el que el cuarto bit sea cero, excepto en el caso en el que el número correspondiente sea 000 y el otro, 111. De esta manera, el 0 pasaría a ser 8 cuando se comparara con el 7. Esta solución es válida porque se supone que no puede haber saltos de dos o más sectores entre dos lecturas consecutivas. El circuito correspondiente a este comparador circular se muestra en la figura 84.

Figura 84. Circuito de un comparador circular de 3 bits




Para la materialización de la EFSM con arquitectura de FSMD se separa la parte de control de la de procesamiento de datos. La unidad de control correspondiente toma las entradas (*A* > *B*), (A = *B*) y (A < *B*) de la unidad de procesamiento, que contiene un comparador circular como el que hemos visto. La unidad de control se ocupa de decidir el estado siguiente de la EFSM y las acciones asociadas a cada estado. En este caso, debe implementar las funciones de transición de estado y las de salida *S*, que ya son la misma salida de la EFSM y que ya se han definido con anterioridad.

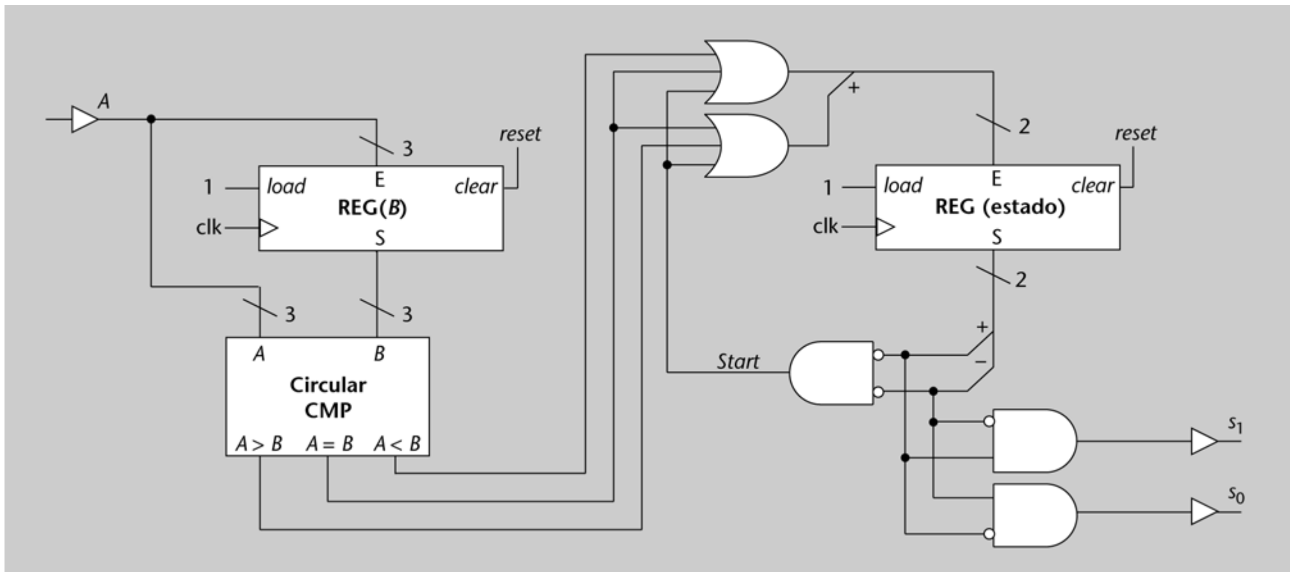
Las funciones de transición se obtienen de la tabla de verdad correspondiente. En este caso, basta con observar que el estado de destino depende exclusivamente de las entradas y que solo hay una entrada activa en cada momento, con la excepción de la transición de salida del estado START.

$$q_1^+ = q_1' \cdot q_0' + (A = B) + (A > B)$$
$$q_0^+ = q_1' \cdot q_0' + (A = B) + (A < B)$$

En la unidad de procesamiento se debe efectuar la operación de carga de la variable $B$ y se han de generar las señales $(A < B)$, $(A = B)$ y $(A > B)$, lo que se lleva a cabo con un comparador circular.

El circuito de tipo FSMD que materializa la EFSM del detector de sentido de giro es el que se muestra en la figura 85.

Figura 85. Circuito secuencial correspondiente al detector de sentido de giro




**3.** Para el diseño de la PSM hay que tener presente que habrá, como mínimo, un estado inicial previo a que el sensor de nivel no haya enviado algún dato en el que no se deberá activar la alarma. En otras palabras, mientras no llegue esta información, el controlador estará apagado o en *off*. En este estado, que se puede denominar OFF, se debe permanecer hasta que no llegue un bit de START por la entrada $s$, es decir, mientras $(s = 0)$.

En el momento en el que $(s = 1)$, se debe realizar la lectura de los cuatro bits siguientes, que contienen el porcentaje de llenado del depósito que ha medido el sensor. Para ello, la máquina pasará a un estado de lectura, READ. En este estado se ejecutará un programa que acumulará la información de los bits de datos en una variable que contendrá, finalmente, el valor del nivel. El primer paso de este programa consiste en inicializar esta variable con 3 bits a cero y el bit más significativo del valor del porcentaje (DATA3 de la figura 64), puesto como valor de unidad, es decir:

$$L^+ = 000s$$

Los otros pasos son similares, ya que consisten en decalar a la izquierda (multiplicar por 2) el valor de $L$ y sumar el bit correspondiente:

$$L^+ = (L << 1) + s$$

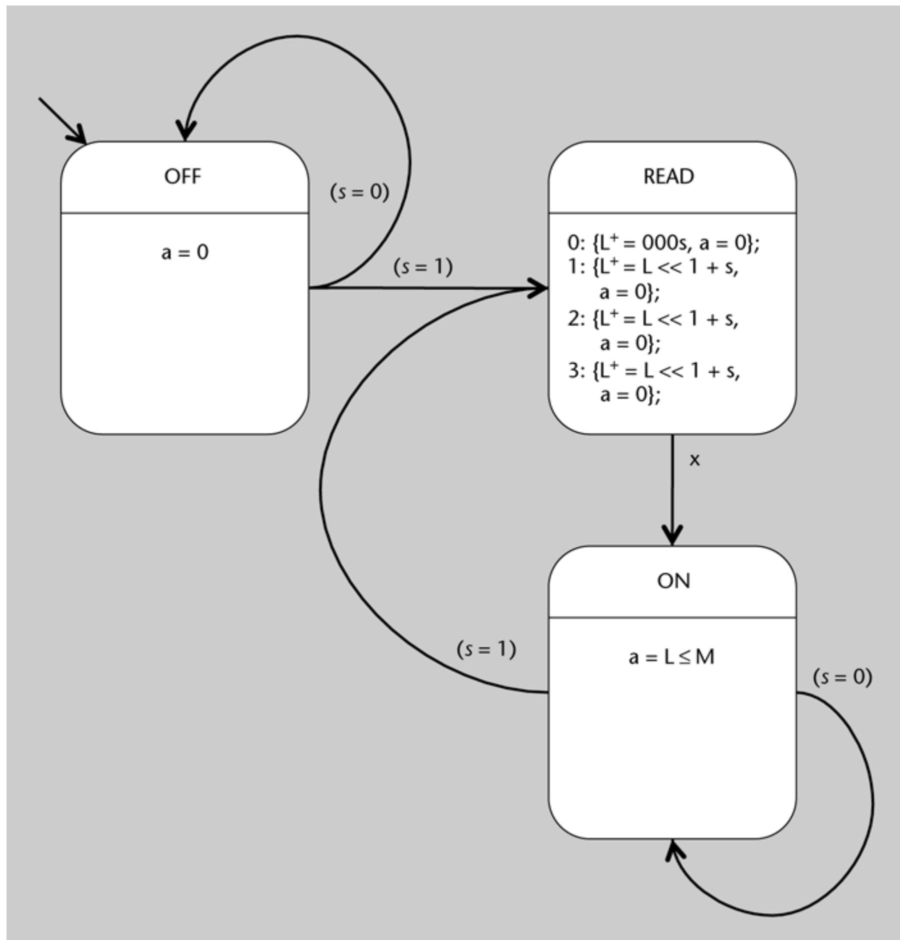
De esta manera, después de 3 pasos, el bit más significativo ya está en la posición más a la izquierda, y el menos significativo, en la posición de las unidades. Por simplicidad, se ha optado por que, durante la ejecución del programa de lectura, la alarma se detenga. En la práctica, esto no debería ser demasiado problemático, dada la frecuencia de trabajo de los circuitos.

Desde el estado-programa de lectura, READ, se debe pasar incondicionalmente a otro estado en el que el controlador esté activo y tenga, como salida, un 1 o un 0, según si $L \leq M$ o no. Por este motivo, se le puede denominar ON. La PSM se debe quedar en este estado hasta que se reciba un nuevo dato, es decir, hasta que $s = 1$.

Como hay cuatro bits de parada a cero (STOP3,..., STOP0), no es posible perder ninguna lectura o hacer una lectura parcial de las series de bits que provienen del sensor. De hecho, solo

es necesario un ciclo de reloj entre el fin del programa de READ y un posible nuevo inicio, lo que evita poner estados intermedios entre READ y ON. En la figura 86 se muestra la PSM resultante.

Figura 86. PSM del controlador de la alarma de aviso de nivel mínimo




De cara a la implementación en un circuito, habría que codificar los tres estados, de manera que OFF fuera 00 y, a modo de ejemplo, READ pudiera ser 01 y ON, 10. También habría que tener un contador interno de dos bits activado con una señal interna, *inc*, que se debería poner a 1 para los valores 0, 1 y 2 del estado-programa READ. La señal de salida $a$ depende exclusivamente del estado en el que se encuentra la PSM:

$$a = \mathrm{ON} \cdot (L \leq M)$$

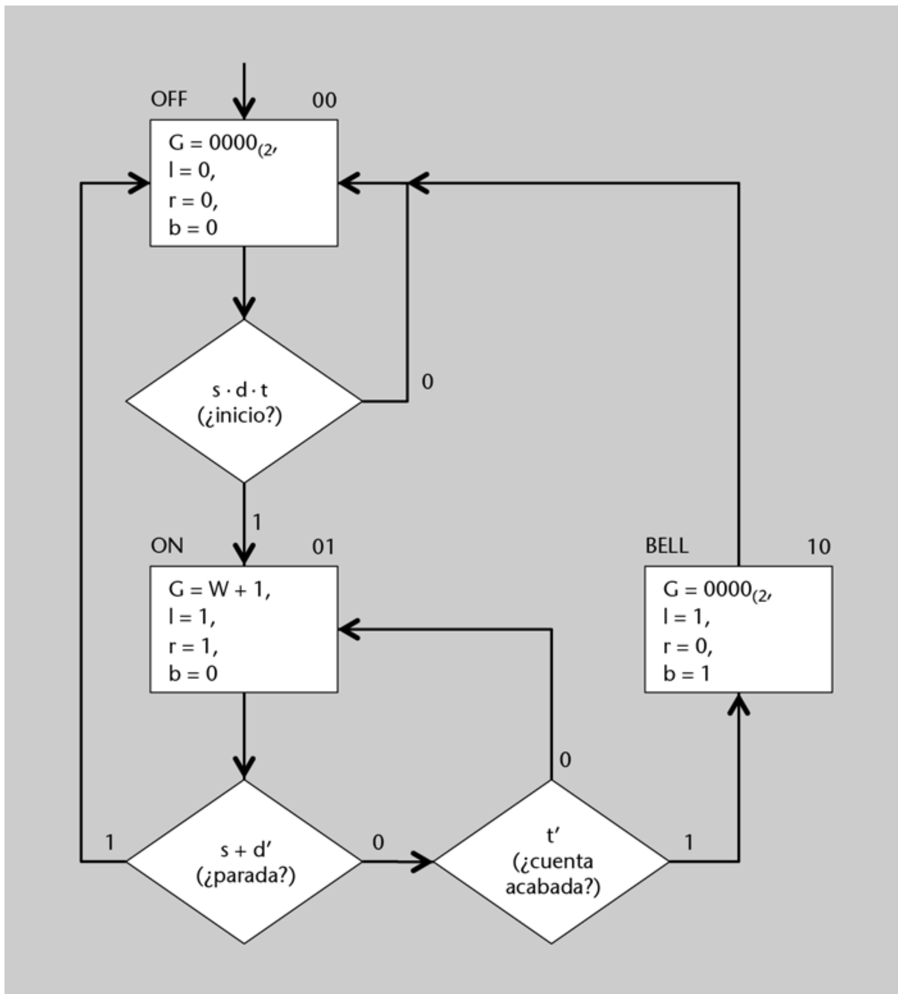
En la unidad de procesamiento debería estar el registro $L$ y los módulos combinacionales necesarios para realizar las operaciones de decalado ($L$ << 1), suma y comparación. Hay que tener presente que se debería seleccionar qué valor se asigna a $L$ con un multiplexor y que, por lo tanto, es necesaria otra salida de la unidad de control, que se puede denominar *sL*, por "selector del valor de entrada de $L$".

**4.** De modo similar al diseño de las máquinas de estados, el procedimiento para diseñar las ASM incluye la elaboración del grafo correspondiente a partir de una caja de estado inicial. En este caso, el horno debe estar apagado (caja de estado OFF). Tal y como se indica en el enunciado, el horno deberá estar en OFF hasta el momento en el que el usuario apriete el botón de inicio/parada ($s$ = 1). En este momento, si la puerta está cerrada ($d$ = 1) y el temporizador está en funcionamiento ($t$ = 1), se puede pasar a un estado de activación del horno (caja de estado ON).

En el estado ON, se debe indicar al generador de microondas la potencia con la que ha de trabajar, que será el valor de la potencia indicada por el usuario (de 0 a 3) incrementado en una unidad. El horno debe permanecer en este estado hasta que el temporizador acabe la cuenta o se interrumpa su funcionamiento, bien al abrir la puerta, bien al apretar el botón de inicio/parada. En los últimos casos, se debe pasar directamente a la caja de estado OFF. En el caso de que se acabe normalmente el funcionamiento porque el temporizador indique la finalización del periodo de tiempo ($t$ = 0), se debe pasar a una caja de estado (BELL) que ayude a crear un pulso a 1 en la salida $b$ para hacer sonar una alarma de aviso. Los valores que se

atribuyen al resto de las salidas en esta caja de estado es relativamente irrelevante: solo estarán activos durante un ciclo de reloj. La ASM correspondiente se puede ver en la figura 87.

Figura 87. ASM del controlador de un horno de microondas




En el esquema de la figura 87 se pueden ver las expresiones lógicas asociadas a cada caja de decisión. En este caso, no se utiliza ninguna variable y, por lo tanto, la relación entre cajas de decisión y de estado es flexible.

En cuanto a la implementación de este controlador, hay que decir que tiene una unidad de procesamiento muy simple, dado que solo hay un cálculo ($W + 1$) y una alternativa de valores para $G$: $0000_{(2}$ o $W + 1$. Así, el problema se reduce a implementar la unidad de control, que, además del cálculo del estado siguiente y de las salidas de bit ($l$, $r$, $b$), debería tener una señal de salida adicional, $sG$, que habría de servir para controlar el valor en la salida $G$.

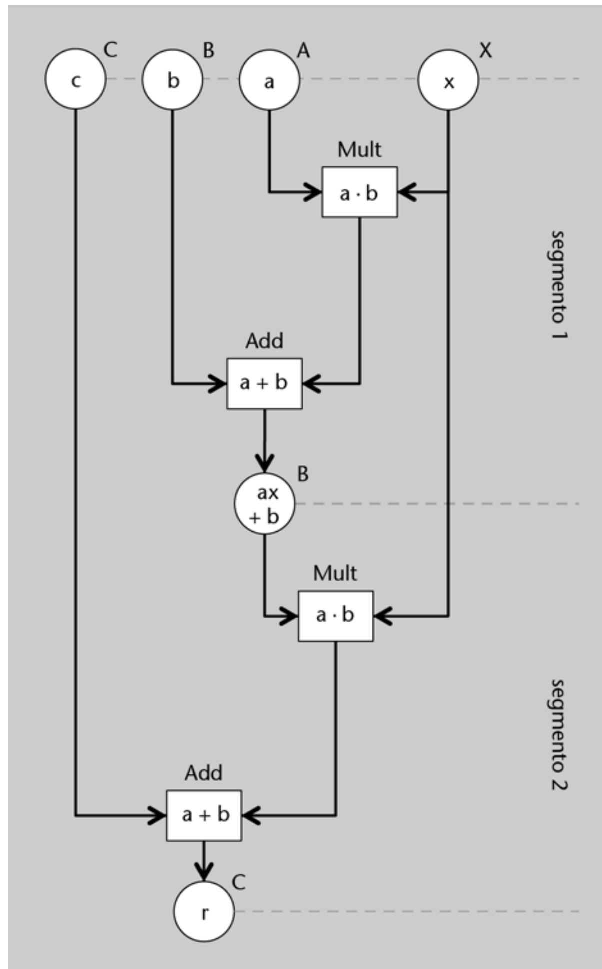
**5.** Como se trata de realizar el cálculo con el mínimo número de recursos, es necesario minimizar el número de operaciones y aprovechar los registros que almacenan las variables de entrada para resultados intermedios y para el final. En cuanto al primer punto, se puede reescribir la expresión en la forma:

$$(ax + b)x + c$$

De este modo, no se hace el producto para calcular $x^2$ y el número de operaciones es mínimo. Basta con hacerlas gradualmente, de manera que solo sea necesaria una de cada tipo en cada ciclo de reloj. Esto irá en contra del número de ciclos de reloj que serán necesarios para realizar el cálculo, que será más elevado.

El resultado es el esquema de cálculo que se muestra en la figura 88, que necesita 2 ciclos de reloj. El esquema se ha etiquetado de manera que se puede ver una posible asignación de cada nodo a los recursos de almacenaje (registros) y de cálculo correspondientes.

Figura 88. Esquema de cálculo para $ax^2 + bx + c$




**6.** En el estado *S*3 se reduce *A* en una unidad porque la actualización de los registros *A* y *P* que se hace en el estado *S*2 tiene efecto al acabar el ciclo en el que se calculan los nuevos valores. Por lo tanto, la condición se hace con los valores que tenían los registros al principio del ciclo, los de la derecha de las expresiones contenidas en el nodo de procesamiento *S*2. Por lo tanto, al llegar a *S*3, *A* se actualiza en *A* + 1, siendo una unidad mayor que la correspondiente al hecho de satisfacer la condición de ser la raíz cuadrada entera de *C*.

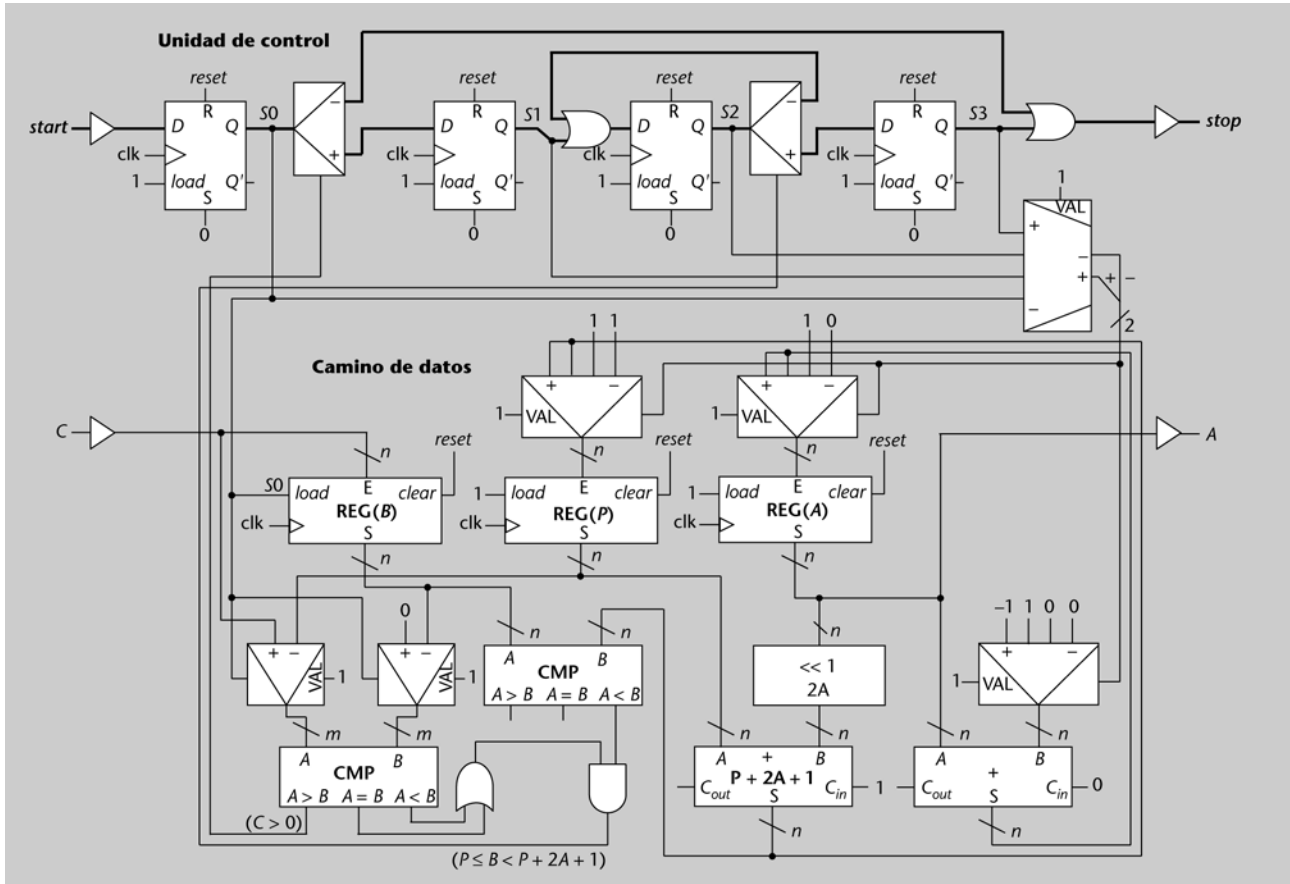
La implementación del circuito se realiza separando las partes de control y de procesamiento, según el modelo de FSMD. La unidad de control se puede materializar con una codificación de tipo *one-hot bit*, lo que permite reproducir directamente el diagrama de flujo del algoritmo correspondiente: los nodos de procesamiento son biestables y los de decisión, demultiplexores.

El camino de datos sigue la arquitectura con multiplexores controlados por el número de estado en la entrada de los recursos, sean de memoria (registros) o de cálculo. En el caso de la solución que se presenta en la figura 89, existen algunas optimizaciones de cara a reducir las dimensiones del esquema del circuito:

- Se ha eliminado el multiplexor en la entrada del registro *B* porque este registro solo carga un valor *S*0 (el contenido de la señal de entrada *C*).
- Se ha reducido el orden de los multiplexores en la entrada del comparador que se ocupa de calcular ($C > 0$) y ($P \leq B$) porque la primera comparación solo se hace en *S*0.
- Se han eliminado los multiplexores del cálculo de $P + 2A + 1$ y de (($P \leq B$) AND ($B < P + 2A + 1$)) porque solo se aprovechan en *S*2.

No obstante, aún existen otras optimizaciones posibles que se dejan como ampliación al ejercicio.

Figura 89. Circuito para el cálculo de la raíz cuadrada entera

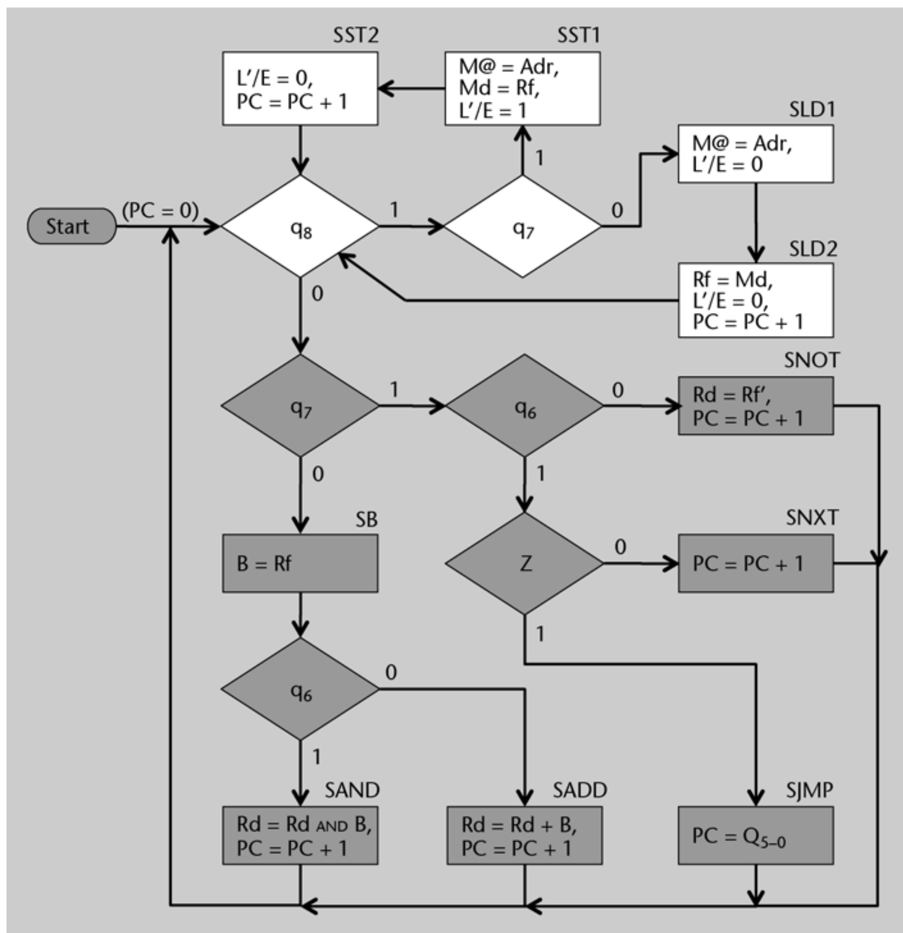



**7.** El algoritmo de interpretación no cambia para las instrucciones ya existentes en el repertorio del Femtoproc, solo hay que añadir la parte correspondiente a la interpretación de las nuevas instrucciones. Para hacerlo, en el diagrama de flujo correspondiente se debe poner un nodo de decisión justo al principio que compruebe el valor del bit más significativo del código de la instrucción ($q_8$). Si es 0, se enlaza con el diagrama anterior, cuyos nodos aparecen sombreados en la figura 90. Si es 1, se pasa a interpretar las nuevas instrucciones.

El bit $q_7$ identifica si la instrucción es de lectura (LOAD) o escritura (STORE) de un dato en memoria. Si se trata de una lectura, es necesario que la memoria reciba la dirección de la posición de memoria que se quiere leer (*M@* = *Adr*, donde *Adr* = $Q_{6-3}$) y que la señal de operación de memoria *L'*/*E* esté a 0. Dado que la memoria necesita un ciclo de reloj para poder obtener el dato de la posición de memoria dada, es necesario un estado adicional (SLD2) en el que ya será posible leer el dato de la memoria y almacenarlo en el registro indicado por las posiciones 2 a 0 del código de instrucción (Rf = *Md*). La salida *L'*/*E* se debe mantener a 0 en todos los estados para evitar hacer escrituras que puedan alterar el contenido de la memoria de manera indeseada, por ello se ha puesto que *L'*/*E* = 0 también en SLD2. En este estado, hay que incrementar el contador de programa para pasar a la instrucción siguiente (*PC* = *PC* + 1).

Si se ejecuta una instrucción STORE, $Q_{8-7} = 11_{(2}$, se debe pasar a un primer estado en el que se indique en la memoria a qué dirección se debe acceder (*M@* = *Adr*) para escribir (*L'*/*E* = 1) y qué contenido se debe poner (*Md* = Rf). El nodo siguiente (SST2) proporciona la espera necesaria para que la memoria haga la escritura y además sirve para calcular la dirección de la instrucción siguiente (*PC* = *PC* + 1) en la memoria de programa. De forma similar a lo que se hace en SLD2, en SST2 se tiene que poner *L'*/*E* a 0. En este sentido, todos los estados del diagrama anterior (SAND, SADD, SJMP, SNOT y SNXT) también tienen que incluir *L'*/*E* = 0, aunque no se muestre en la figura 90 por motivos de legibilidad.

Figura 90. Diagrama de flujo del Femtoproc ampliado




**8.** En el camino de datos, el registro de direcciones de memoria (MAR) debería tener 16 bits. Para no modificar nada más, podría añadirse un segundo registro de 8 bits, igualmente conectado al *BusW*. Así, desde el punto de vista del camino de datos, una dirección está formada por dos bytes: el menos significativo (el byte bajo o *low byte*) se debería almacenar en el registro $MAR_L$ (el actual MAR) y el más significativo, en el nuevo registro $MAR_H$. Desde el punto de vista de la memoria, los dos registros ($MAR_L$ y $MAR_H$) se verían como una unidad, que le proporcionaría una dirección de 16 bits.

La unidad de control debería implementar un ciclo de ejecución de instrucciones en el que las que trabajan con direcciones completas de memoria necesitarían una lectura adicional para leer el segundo byte de la dirección. En el caso del modo indirecto, serían dos lecturas adicionales, ya que hay que leer una segunda dirección de memoria. Por lo tanto, la ejecución de las instrucciones afectadas sería más lenta. (Una solución sería aumentar el número de registros del camino de datos para poder reducir el número de accesos a datos en memoria.)

# Glosario

**acceso directo a memoria** *m* Mecanismo que permite a los dispositivos periféricos de un computador acceder directamente a la memoria principal del procesador.
*en* direct memory access
sigla **DMA**

**algoritmo** *m* Método para resolver un problema que se especifica como una secuencia de pasos finita. Cada paso consiste o en una decisión o en un proceso que, a su vez, puede ser otro algoritmo.

***algorithmic state-machine*** *f* Véase **máquina de estados algorítmica**.

**ALU** *f* Véase **unidad aritmeticológica**.

**arquitectura** *f* Modelo de construcción que describe los componentes (recursos) de un sistema y las relaciones que deben tener.

**arquitectura Hardvard** *f* Arquitectura de un procesador en el que las instrucciones de un programa se guardan en una memoria diferenciada de la de los datos que manipula.

**arquitectura del repertorio de instrucciones** *f* Modelo de codificación de las instrucciones en un repertorio dado.
*en* instruction-set architecture
sigla **ISA**

**arquitectura de Von Neumann** *f* Arquitectura de un procesador en la que el programa es almacenado en la memoria.

**ASM** *f* Véase **máquina de estados algorítmica**.

**bus** *m* Sistema de comunicación entre dos o más componentes de un computador constituido por un canal de comunicación (líneas de interconexión), un protocolo de transmisión (emisión y recepción) de datos y módulos que lo materializan.

**bus de direcciones** *m* Parte de un bus dedicada a la identificación de la ubicación de los datos que se transmiten en el dispositivo o dispositivos receptores.

**bus de control** *m* Parte de un bus que contiene señales de control de la transmisión o, si se quiere, de las señales para la ejecución de los protocolos de intercambio de información.

**bus de datos** *m* Parte de un bus que se ocupa de la transmisión de los datos.

**camino de datos** *m* Circuito compuesto por un conjunto de recursos de memoria, que almacenan los datos, de recursos de cálculo, que hacen operaciones con los datos, y de interconexiones, que permiten que los datos se transmitan de un componente a otro. Se denomina así porque el procesamiento de los datos se efectúa de manera que los datos siguen un determinado camino desde las entradas hasta convertirse en resultados en las salidas.

***central processing unit*** *f* Véase **unidad central de procesamiento**.

**computador** *m* Máquina capaz de procesar información de manera automática. Para ello, dispone, al menos, de un procesador y de varios dispositivos de entrada y salida de información.

**controlador** *m* Circuito que puede actuar sobre otra entidad (habitualmente, otro circuito) mediante cambios en las señales de salida. Normalmente, estos cambios varían en función del estado interno y de las señales de entrada, que suelen incluir información sobre la entidad que controlan.

***core*** *m* Véase **núcleo**.

**CPU** *f* Véase **unidad central de procesamiento**.

**diagrama de flujo** *f* Esquema gráfico con varios elementos que representan las distintas sucesiones de acciones (y, por lo tanto, de estados) posibles de un determinado algoritmo.

***digital-signal processor*** *m* Véase **procesador digital de señal**.

***direct memory access*** *m* Véase **acceso directo a memoria**.

**dispositivo periférico** *m* Cualquier componente de un computador que no se incluya dentro del procesador.

**dispositivo periférico de entrada** *m* Componente de un computador que permite introducir información. Habitualmente, está formado por una parte externa al propio computador (teclado, pantalla táctil, ratón, etc.) y otra más interna (el controlador del dispositivo o el módulo de entrada correspondiente), aunque periférica al procesador.

**dispositivo periférico de entrada/salida** *m* Componente de un computador que le permite introducir y extraer información. Normalmente, es una memoria secundaria a la del procesador y presenta una arquitectura con dos partes: una externa, como unidades de disco duro u óptico o como los conectores de los módulos para memorias USB, y otra interna, que incluye el controlador del dispositivo o el módulo de entrada/salida correspondiente, pero fuera del procesador.

**dispositivo periférico de salida** *m* Componente de un computador que permite extraer información. Normalmente, la información es observable en una parte externa del propio

computador (pantalla, impresora, altavoz, etc.), aunque hay otra más interna (el controlador del dispositivo o el módulo de salida correspondiente), aunque es periférica al procesador.

**DMA** *m* Véase **acceso directo a memoria**.

**DSP** m Véase **procesador digital de señal**.

**EFSM** *f* Véase **máquina de estados finitos extendida**.

**esquema de cálculo** *m* Representación gráfica de un cálculo en función de los operandos y operadores que contiene.

**estructura** *f* Conjunto de componentes de un sistema y cómo están relacionados.

***extended finite-state machine*** *f* Véase **máquina de estados finitos extendida**.

***finite-state machine*** *f* Véase **máquina de estados finitos**.

***finite-state machine with data-path*** *f* Véase **máquina de estados finitos con camino de datos**.

**FSM** *f* Véase **máquina de estados finitos**.

**FSMD** *f* Véase **máquina de estados finitos con camino de datos**.

**GPU** *f* Véase **unidad de procesamiento de gráficos**.

***graphics processing unit*** *f* Véase **unidad de procesamiento de gráficos**.

**Hardvard (arquitectura)** *f* Véase **arquitectura Hardvard**.

***instruction-set architecture*** *f* Véase **arquitectura del repertorio de instrucciones**.

**ISA** *f* Véase **arquitectura del repertorio de instrucciones**.

**lenguaje máquina** *m* Lenguaje binario inteligible para una máquina. Normalmente, consiste en una serie de palabras binarias que codifican las instrucciones de un programa.

**máquina algorítmica** *f* Modelo de materialización del hardware correspondiente a un algoritmo, habitualmente representado por un diagrama de flujo.

**máquina de estados algorítmica** *f* Modelo de representación del comportamiento de un circuito secuencial con elementos diferenciados para los estados y para las transiciones. Permite trabajar con sistemas con un gran número de entradas.
*en* algorithmic state-machine
sigla **ASM**

**máquina de estados finitos** *f* Modelo de representación de un comportamiento basado en un conjunto finito de estados y de las transiciones que se definen para pasar de uno al otro.
*en* finite-state machine
sigla **FSM**

**máquina de estados finitos extendida** *f* Modelo de representación de comportamientos que consiste en una máquina de estados finitos que incluye cálculos con datos tanto en las transiciones como en las salidas asociadas a cada estado.
*en* extended finite-state machine
sigla **EFSM**

**máquina de estados finitos con camino de datos** *f* Arquitectura que separa la parte de cálculo de la de control, que es una máquina de estados finitos definida en términos de funciones lógicas para las transiciones y las salidas asociadas a cada estado.
*en* finite-state machine with data-path
sigla **FSMD**

**máquina de estados-programa** *f* Modelo de representación de comportamientos de sistemas secuenciales en los que cada estado puede implicar la ejecución de un programa.
*en* program-state machine
sigla **PSM**

**MCU** *m* Véase **microcontrolador**.

**memoria caché** *f* Memoria interpuesta en el canal de comunicación de dos componentes para hacer más efectiva la transmisión de datos. Normalmente, la hay entre CPU y memoria principal, y entre procesador y periféricos. En el primer caso, puede haber varias interpuestas en cascada para una adaptación más progresiva de los parámetros de velocidad de trabajo y de capacidad de memoria.
*en* cache memory

**memoria principal** *f* Memoria del procesador.

**microarquitectura** *f* Arquitectura de un determinado procesador.

**microcontrolador** *m* Procesador con una microarquitectura específica para ejecutar programas de control. Normalmente, se trata de procesadores con módulos de entrada/salida de datos variados y numerosos.
*en* micro-controller unit
sigla **MCU**

***micro-controller unit*** *m* Véase **microcontrolador**.

**microinstrucción** *f* Cada una de las posibles instrucciones en un microprograma.

**microprograma** *m* Programa que ejecuta la unidad de control de un procesador.

**núcleo** *m* En computadores, un núcleo (de un procesador) es un bloque con capacidad de ejecutar un proceso. Habitualmente se corresponde con una CPU.
*en* core

**periférico** *m* Véase **dispositivo periférico**.

***pipeline*** *m* Encadenamiento en cascada de varios segmentos de un mismo cálculo para poder hacerlo en paralelo.

**procesador** *m* Elemento capaz de procesar información.

**procesador digital de señal** *m* Procesador construido con una microarquitectura específica para el procesamiento intensivo de datos.
*en* digital-signal processor
sigla **DSP**

**programa** *m* Conjunto de acciones ejecutadas en secuencia.

***program-state machine*** *f* Véase **máquina de estados-programa**.

**PSM** *f* Véase **máquina de estados-programa**.

**segmento** *m* Parte de un cálculo que se lleva a cabo en un mismo periodo en una cascada de cálculos parciales que lleva a un determinado cálculo final.

**secuenciador** *m* Máquina algorítmica que se ocupa de ejecutar microinstrucciones en secuencia, según un microprograma determinado.

**unidad aritmeticológica** *f* Recurso de cálculo programable que hace tanto operaciones de tipo aritmético, como la suma y la resta, como de tipo lógico, como el producto (conjunción) y la suma (disyunción) lógicas.
*en* arithmetic logic unit
sigla **ALU**

**unidad central de procesamiento** *m* Parte de un procesador que hace el procesamiento de la información que tiene una microarquitectura con memoria segregada. Esta unidad normalmente tiene una arquitectura de FSMD.
*en* central processing unit
sigla **CPU**

**unidad de control** *f* Parte de un circuito secuencial que controla las operaciones. Habitualmente, se ocupa de calcular el estado siguiente de la máquina de estados que materializa y las señales de control para el resto del circuito.

**unidad de proceso** *f* Parte de un circuito secuencial que se ocupa del procesamiento de los datos. Habitualmente, es la parte que hace las operaciones con los datos tanto para determinar condiciones de transición de la parte de control como resultados de cálculos de salida del circuito en conjunto.
sin. **unidad operacional**

**unidad de procesamiento de gráficos** *f* DSP adaptado al procesamiento de gráficos. Normalmente, con microarquitectura paralela.
*en* graphics processing unit
sigla **GPU**

**unidad operacional** *f* Véase **unidad de proceso**.

**variable** *f* Elemento de los modelos de máquinas de estados que almacena información complementaria en los estados. Normalmente, hay una variable por dato no directamente relacionada con el estado y cada variable se asocia con un registro a la hora de materializar la máquina correspondiente.

**Von Neumann (arquitectura de)** *f* Véase **arquitectura de Von Neumann**.

# Bibliografía

**Lloris Ruiz, A.; Prieto Espinosa, A.; Parrilla Roure, L.** (2003). *Sistemas Digitales*. Madrid: McGraw-Hill.

**Morris Mano, M.** (2003). *Diseño Digital*. Madrid: Pearson-Education.

**Ribas i Xirgo, Ll.** (2000). *Pràctiques de fonaments de computadors*. Bellaterra (Cerdanyola del Vallès): Servei de publicacions de la UAB.

**Ribas i Xirgo, Ll. y otros** (2010). “La robótica como elemento motivador para un proyecto de asignatura en Fundamentos de Computadores”. *Actas de las XVI Jornadas de Enseñanza Universitaria de la Informática* (JENUI) (pág. 171-178). Santiago de Compostela.

**Ribas i Xirgo, Ll.** (2010). “Yet Another Simple Processor (YASP) for Introductory Courses on Computer Architecture". *IEEE Trans. on Industrial Electronics* (núm. 10, vol. 57, octubre, 3317-3323). Piscataway (Nueva Jersey): IEEE.

**Roth, Jr., Ch. H.** (2004). *Fundamentos de diseño lógico*. Madrid: Thomson-Paraninfo.