Estructura básica de un computador

El procesador como generalización de las máquinas algorítmicas

Lluís Ribas i Xirgo

PID_00215621

UOC
Universitat Oberta de Catalunya

Índice

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.

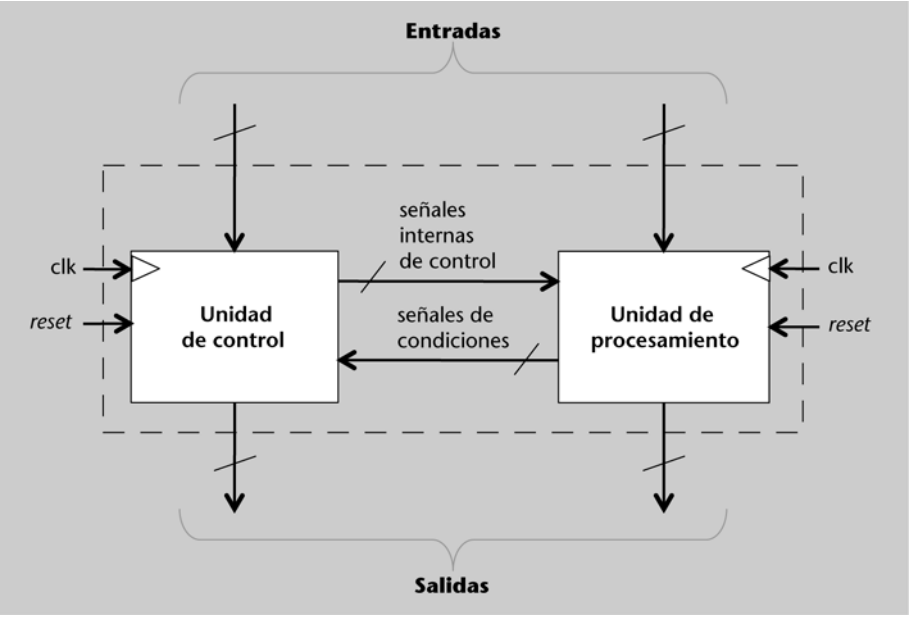
1. Máquinas de estados

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).

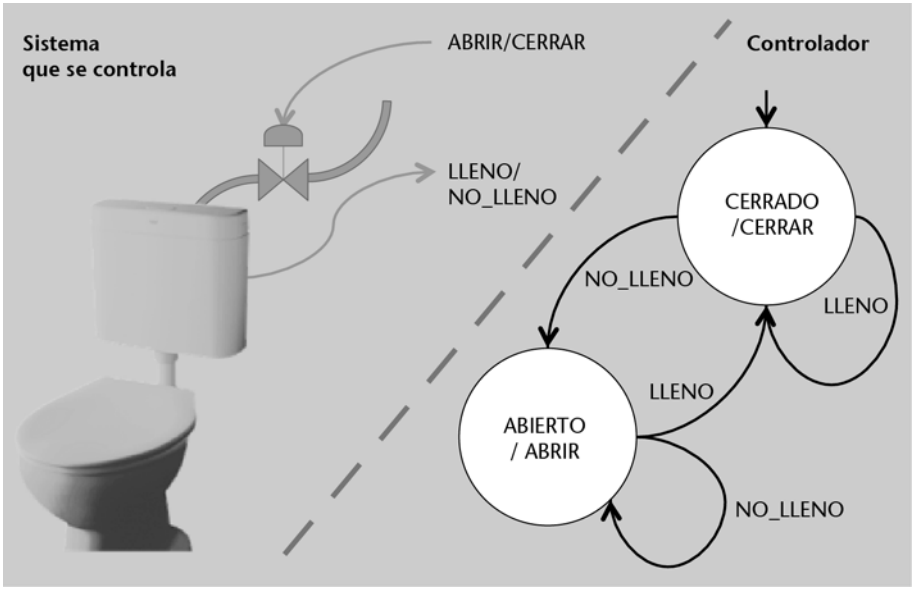
El mecanismo de control de llenado de agua se ocupa de llevar el sistema al estado de LLENO de manera autónoma. En otras palabras, el controlador tiene una referencia interna del estado al que ha de llevar el sistema que controla. Para hacerlo, debe detectar el estado y, con este dato de entrada, determinar 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**.

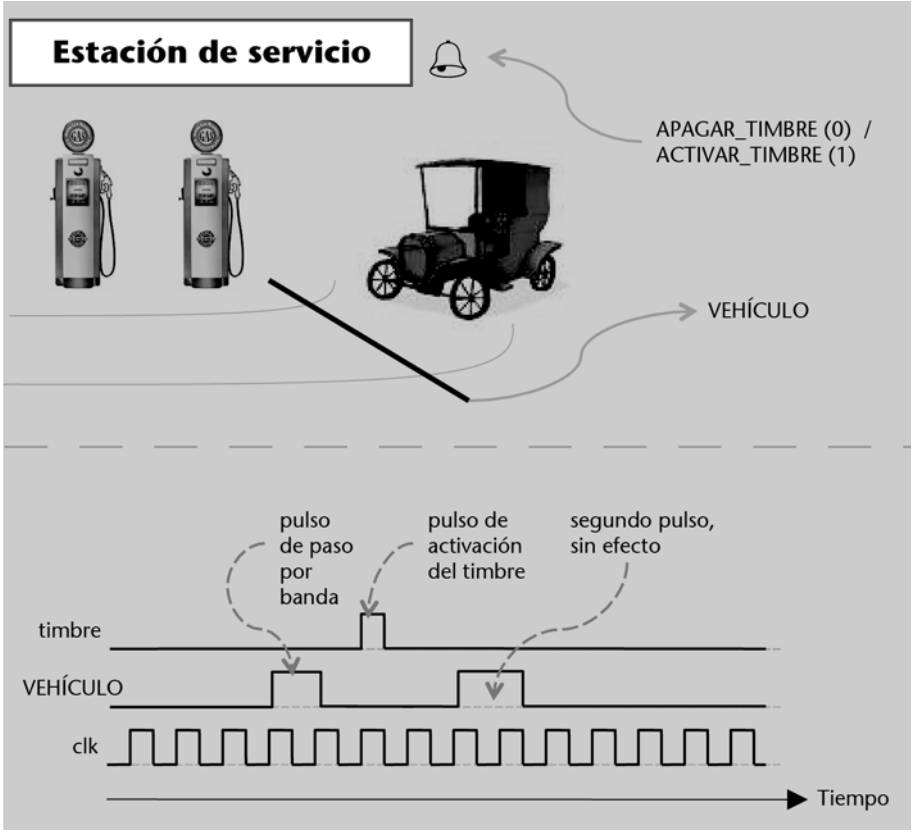
1.1.1. Procedimiento de materialización de controladores con circuitos secuenciales

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.

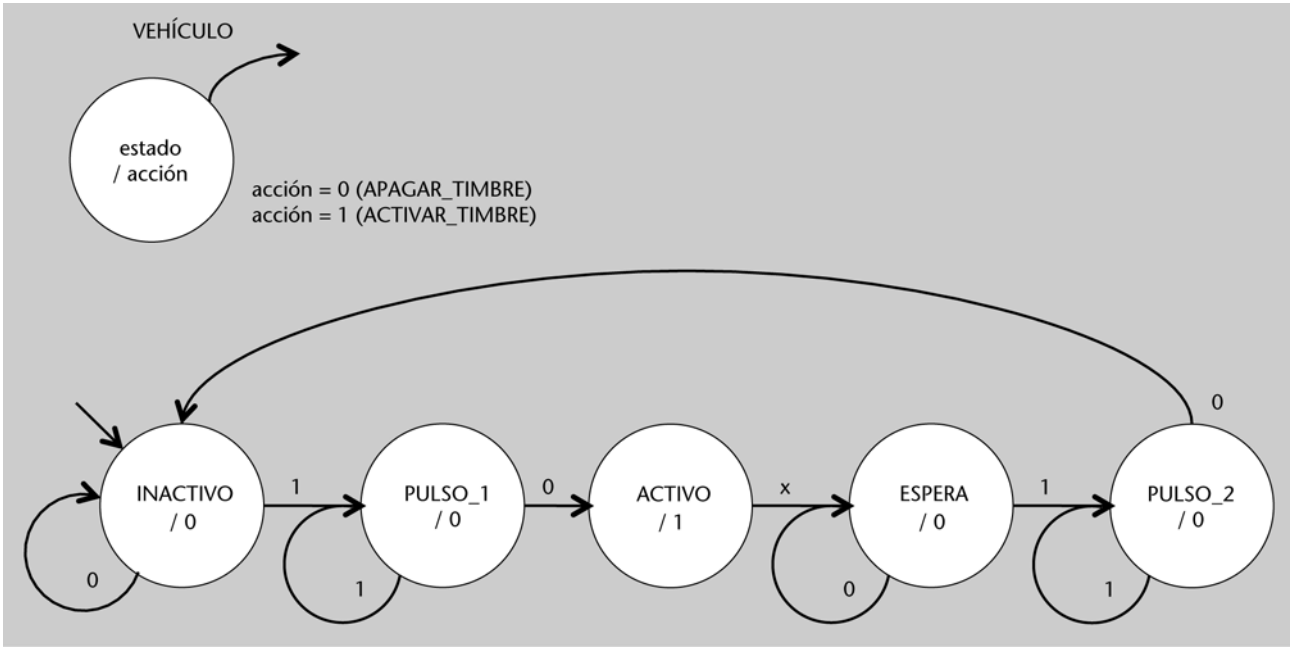
Hay que tener presente que la velocidad de muestreo de la señal VEHÍCULO, es decir, el número de lecturas de este bit por unidad de tiempo, es la del reloj del circuito: mucho mayor que la velocidad de paso de cualquier vehículo. Por lo tanto, la detección de un pulso se realiza al completar una secuencia 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.

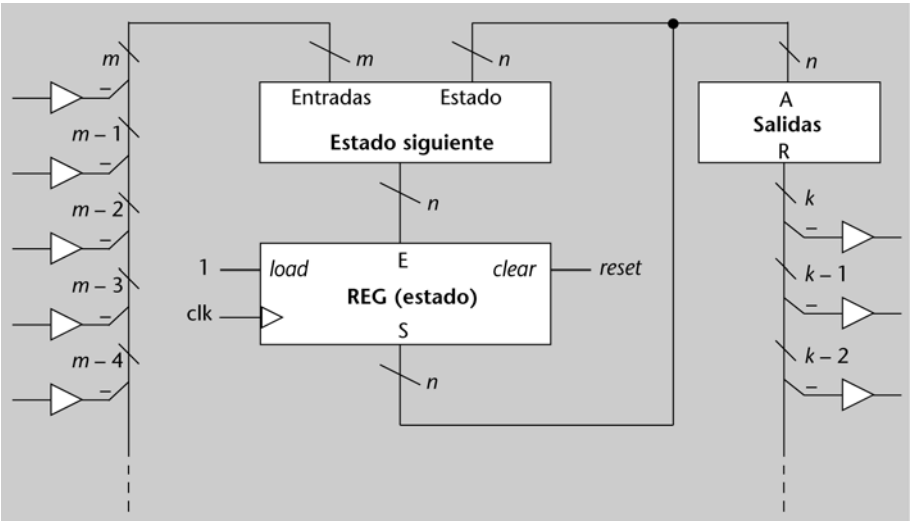
5) Diseñar los circuitos correspondientes.

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

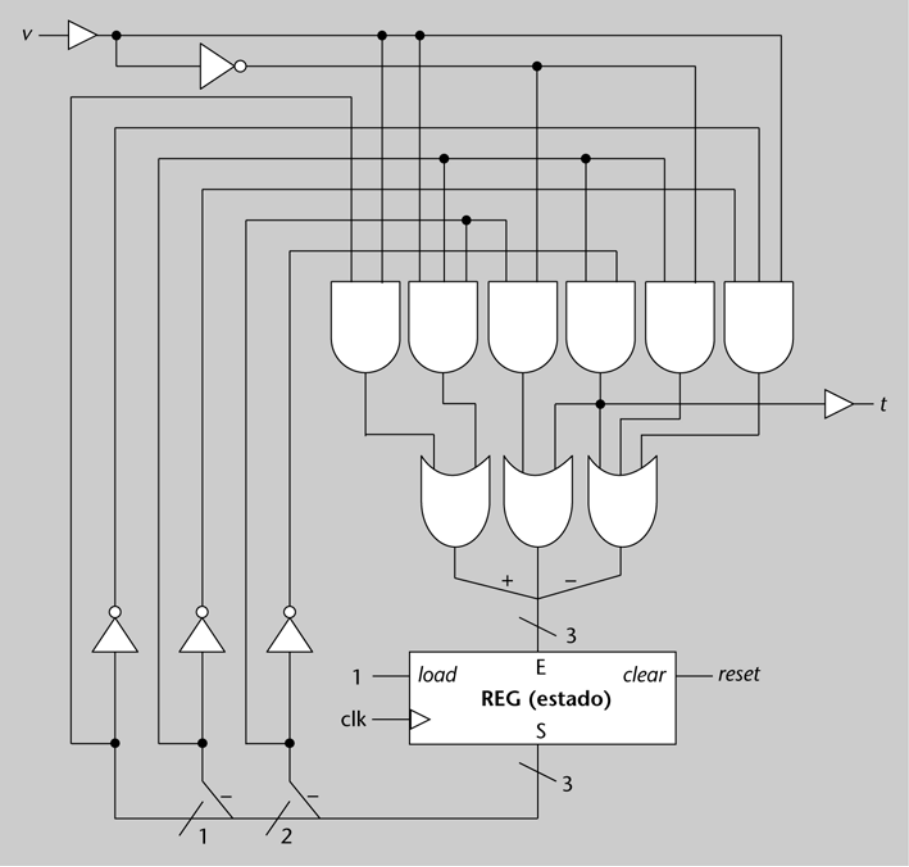


Aunque la parte combinacional se separa en dos: la que calcula el estado siguiente y la que calcula las señales de salida, puede haber elementos comunes que pueden ser compartidos.

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 ($q_1 q_0'$) 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).

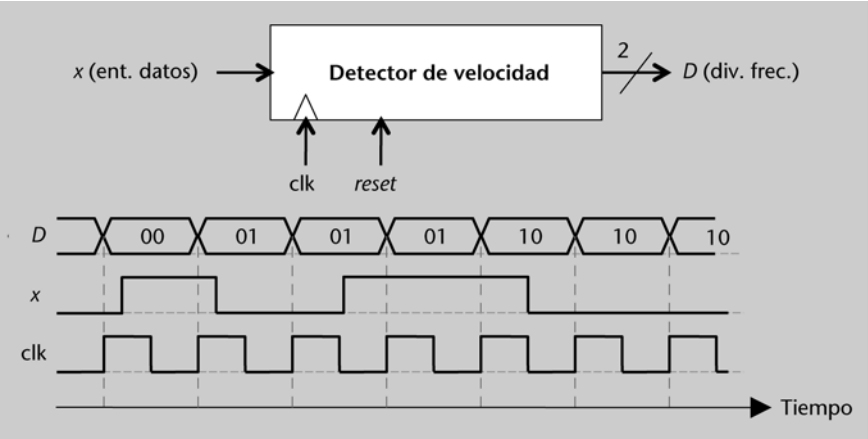
Si un pulso de entrada supera los tres ciclos de reloj, la salida también será 11.

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.

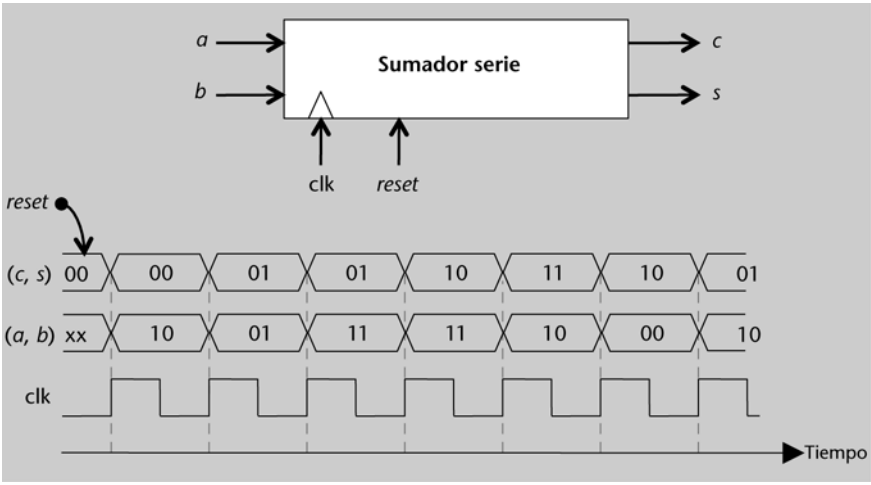
Figura 7. Esquema del detector de velocidad de transmisión



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”.

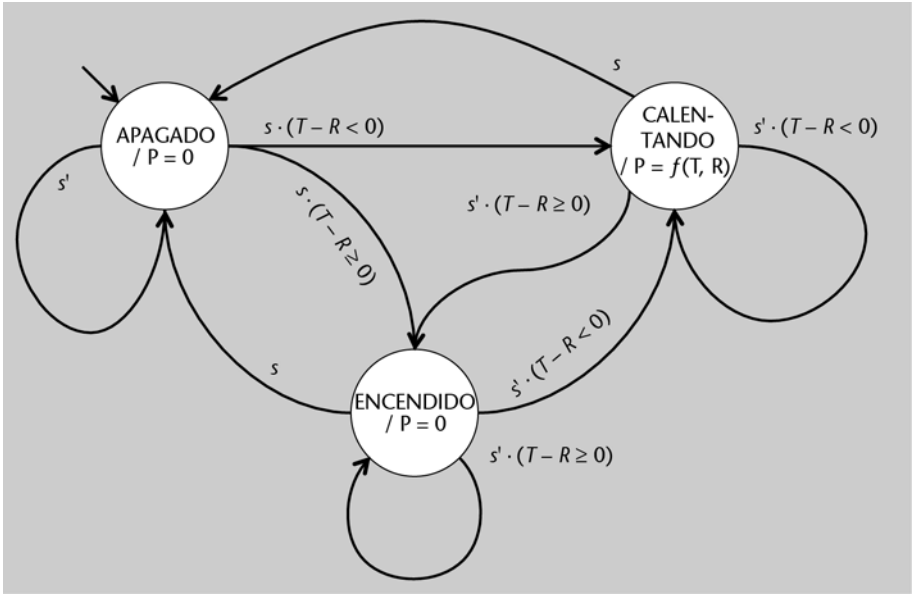
Con todo, es más conveniente incluir en los grafos de estados no solo los bits de entrada y los de salida, sino también las evaluaciones de condiciones sobre valores de entrada y los cálculos para obtener los valores de salida de una manera directa, sin tener que hacer ninguna traducción inicial a valores 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).

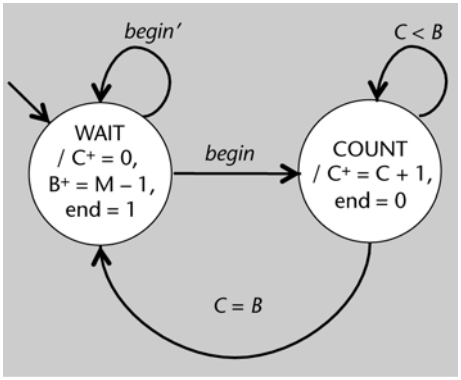
De esta manera, se puede hacer un modelo de EFSM para un contador autónomo que tenga un único estado en el que la acción consista en cargar el contenido de hacer el cálculo de C^+ al registro correspondiente.

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:

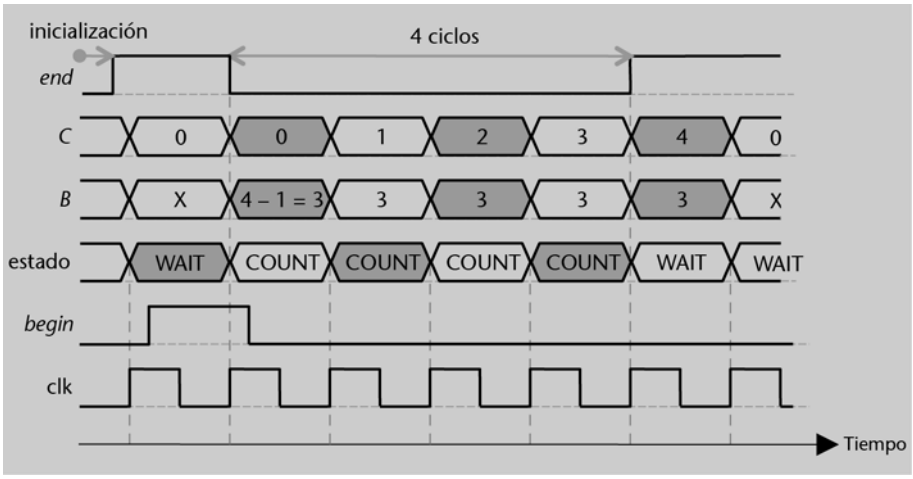
1) Para las señales de salida, se expresa el valor que deben tener en aquel estado. Por ejemplo, al llegar al estado COUNT, *end* será 0.

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.

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".

- 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.

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.

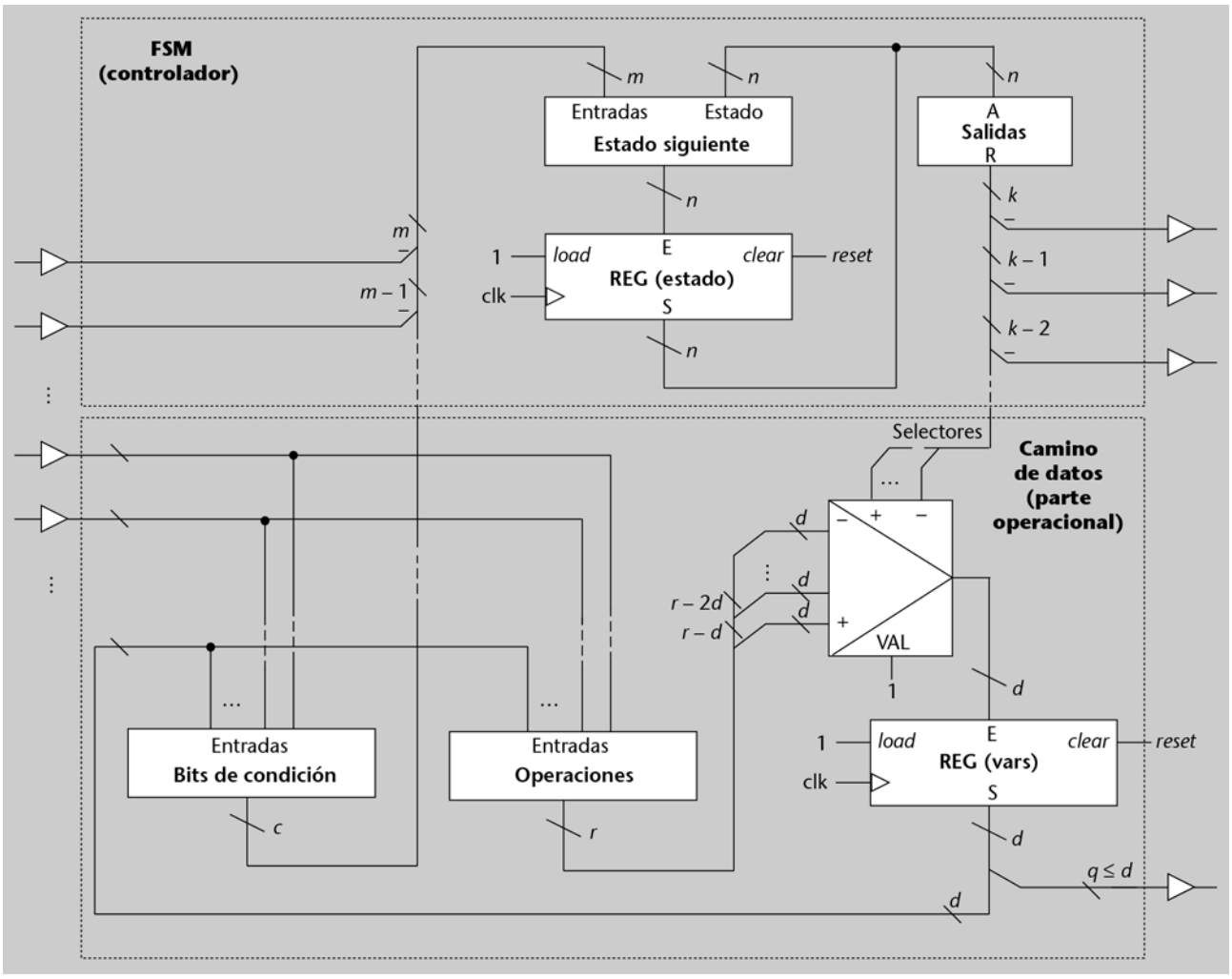
Estado actual		Entradas			Estado$^+$
Símbolo	*s*	*begin*	*C < B*	*C = B*	s^+
WAIT	0	0	x	x	0
WAIT	0	1	x	x	1
COUNT	1	x	0	1	0
COUNT	1	x	1	0	1

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.

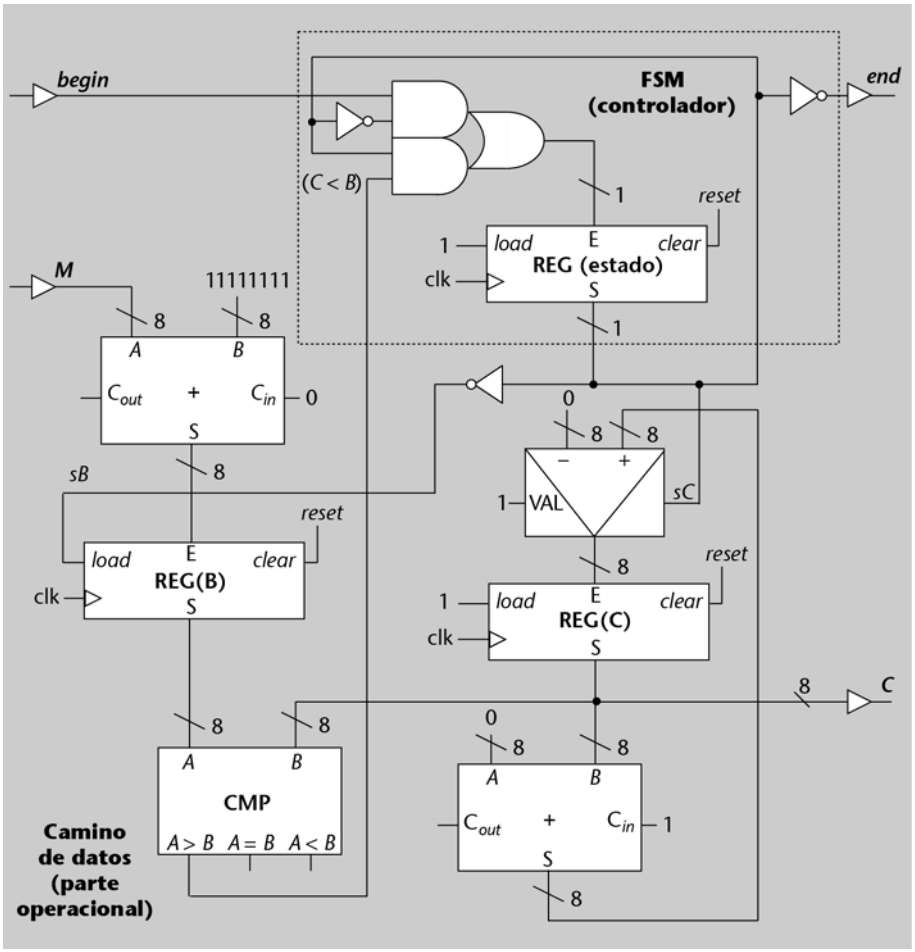
Esta arquitectura, que separa el circuito secuencial en dos unidades: la de control y la de procesamiento, se denomina **arquitectura de máquina de estados con camino de datos** o **FSMD**, que es el acrónimo del inglés *finite-state machine with datapath*.

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 $s^+ = s' \cdot \mathit{begin} + s \cdot (C < B)$, 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.

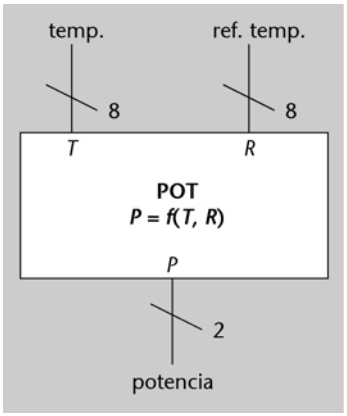
Figura 13. Circuito secuencial correspondiente al contador del ejemplo



Actividades

3. Implementad el circuito de la EFSM del termostato. Para ello, se dispone de un módulo, POT, que calcula $f(T, R)$.

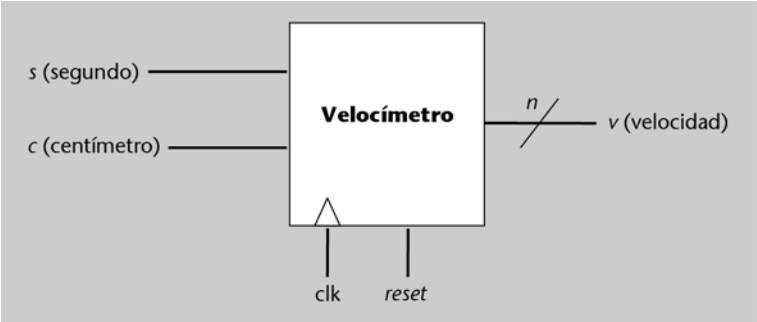
Figura 14. Esquema de entradas y salidas del módulo POT



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.

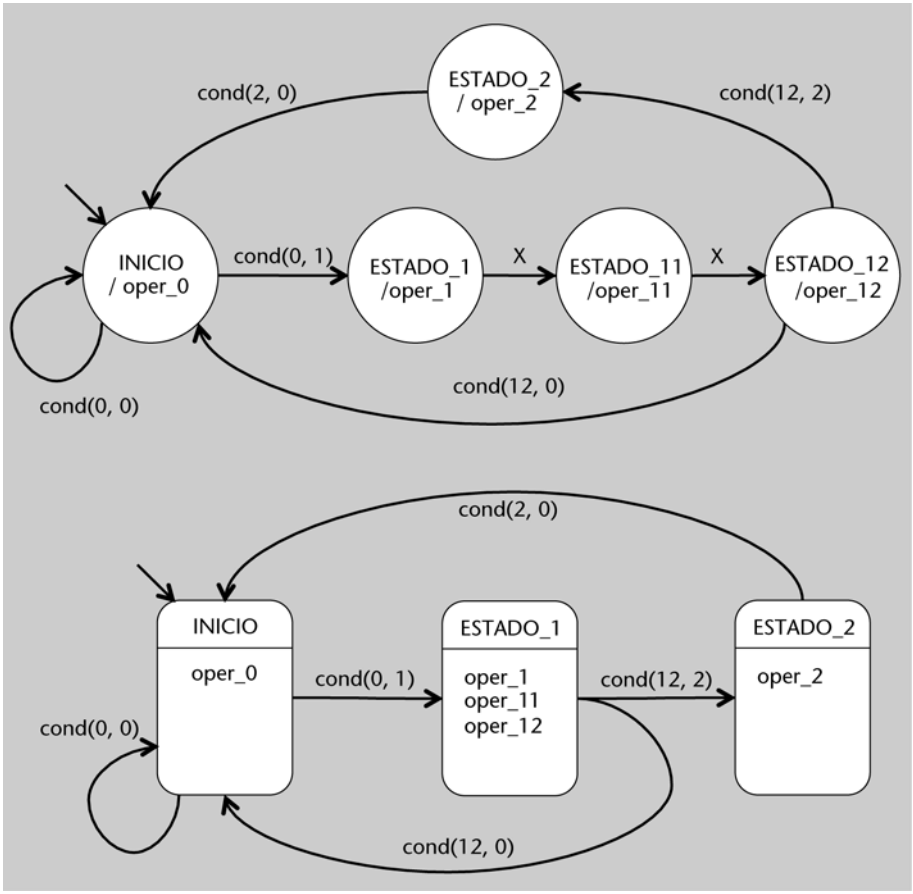
Los programas de cada estado son, de hecho, un conjunto de acciones que hay que ejecutar en secuencia. (En el modelo general de PSM, los programas no tienen ninguna restricción, pero en este texto se supondrá que son series de acciones que se llevan a cabo una tras otra, sin saltos).

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:

$$\{S^+ = V_1,\ y = M \geq T\};$$
$$\{S^+ = S + V_2,\ y = M \geq T\};$$
$$\{S^+ = S + V_3,\ y = M \geq T\};$$
$$\{S^+ = S + V_4,\ y = M \geq T\};$$
$$\{M^+ = S/4,\ y = S/4 \geq T\}$$

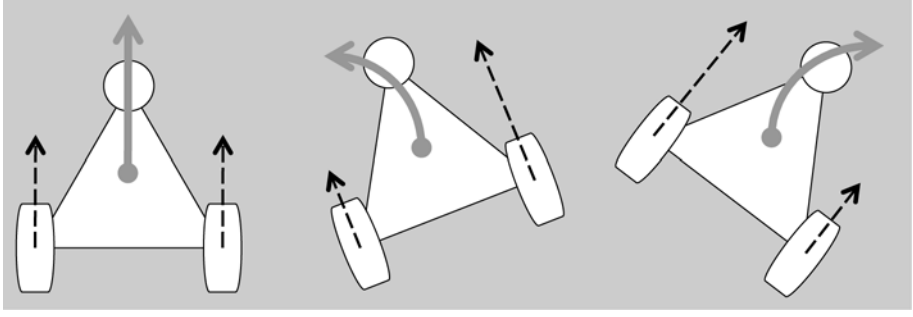
Este programa utiliza, además, dos variables para contener la suma (S) y la media (M) de los valores de entrada. Hay que tener presente que:

- 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"



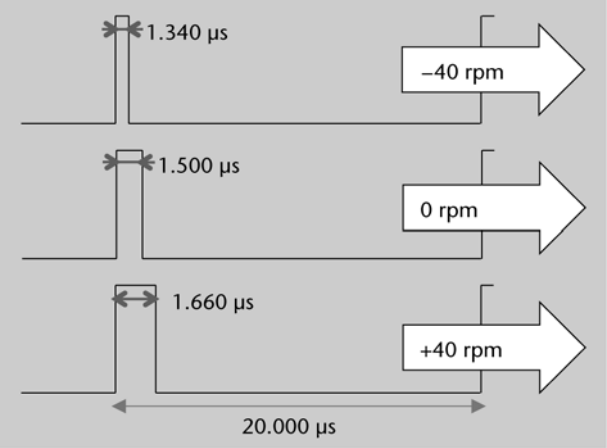
¿Cómo funcionan los servomotores?

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



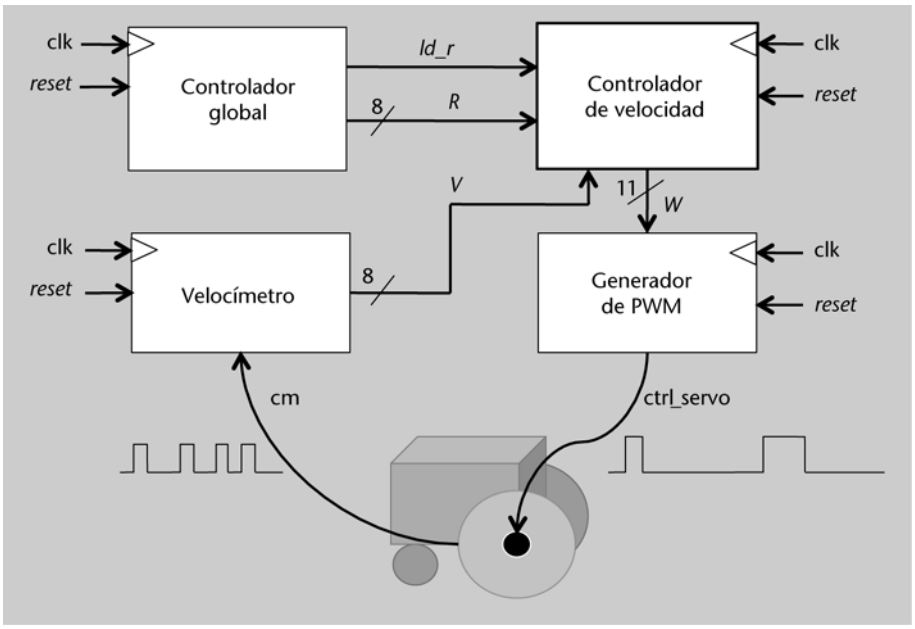
Desgraciadamente, la relación entre el ancho de pulso y la velocidad en revoluciones por minuto (rpm) no es lineal (podéis ver la figura 20) y puede cambiar a lo largo del tiempo.

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.

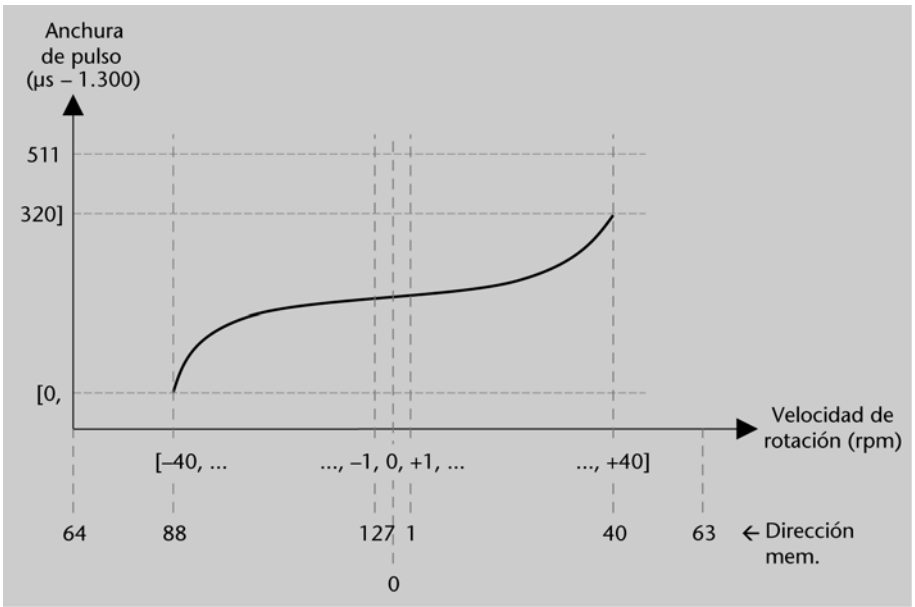
Figura 19. Esquema de los módulos de un controlador de un robot



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*:

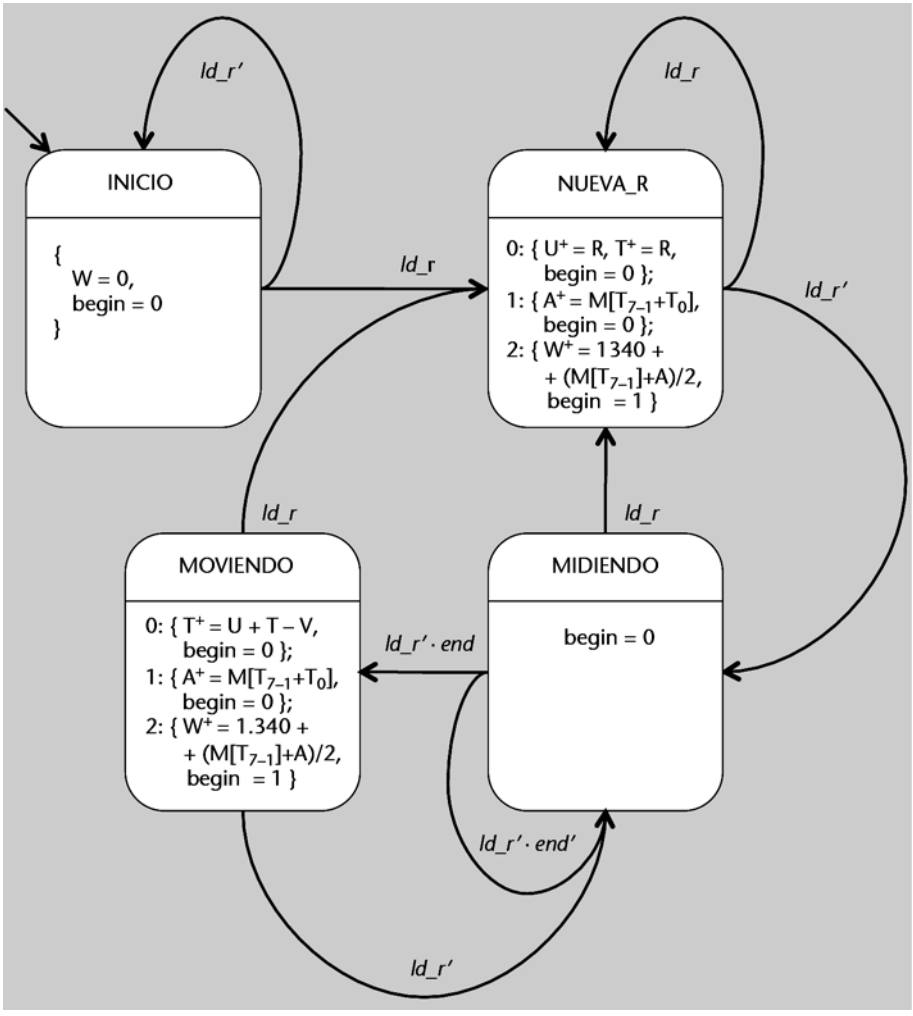
$$T^+ = U + (T - V)$$

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



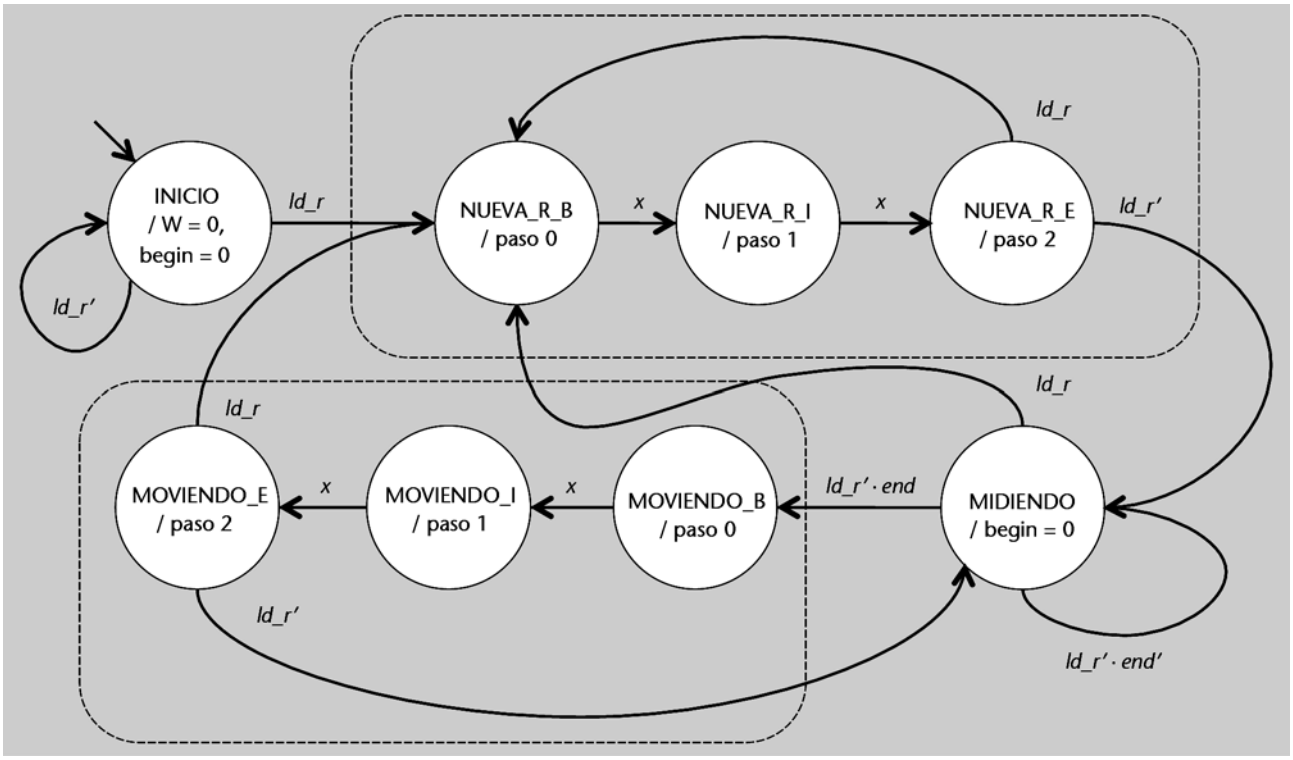
La construcción de las PSM se puede realizar de dos maneras:

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



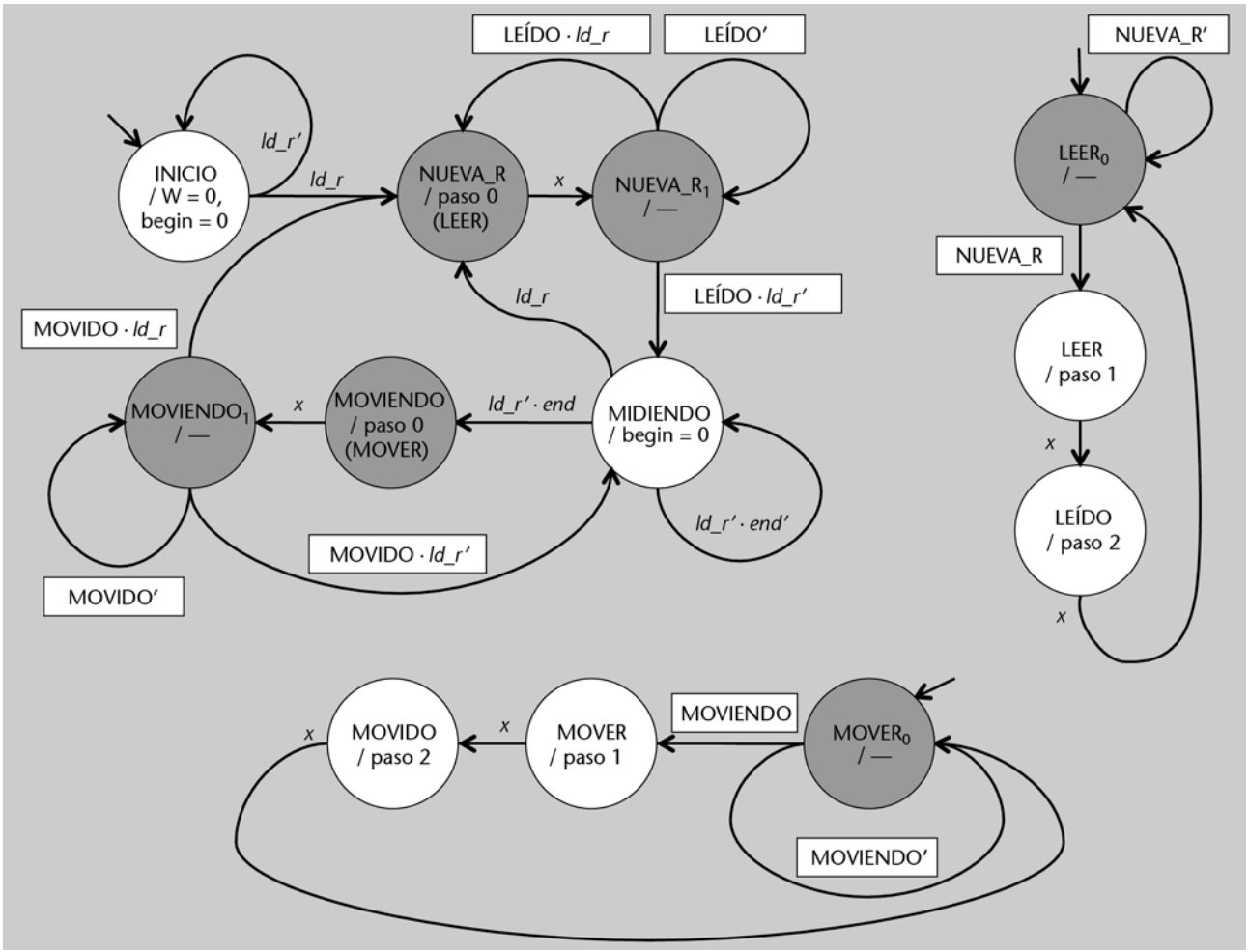
En el segundo caso, el diseño de las EFSM es mucho más simple, a cambio, como se verá, de tener que añadir estados adicionales y señales para coordinarlas.

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



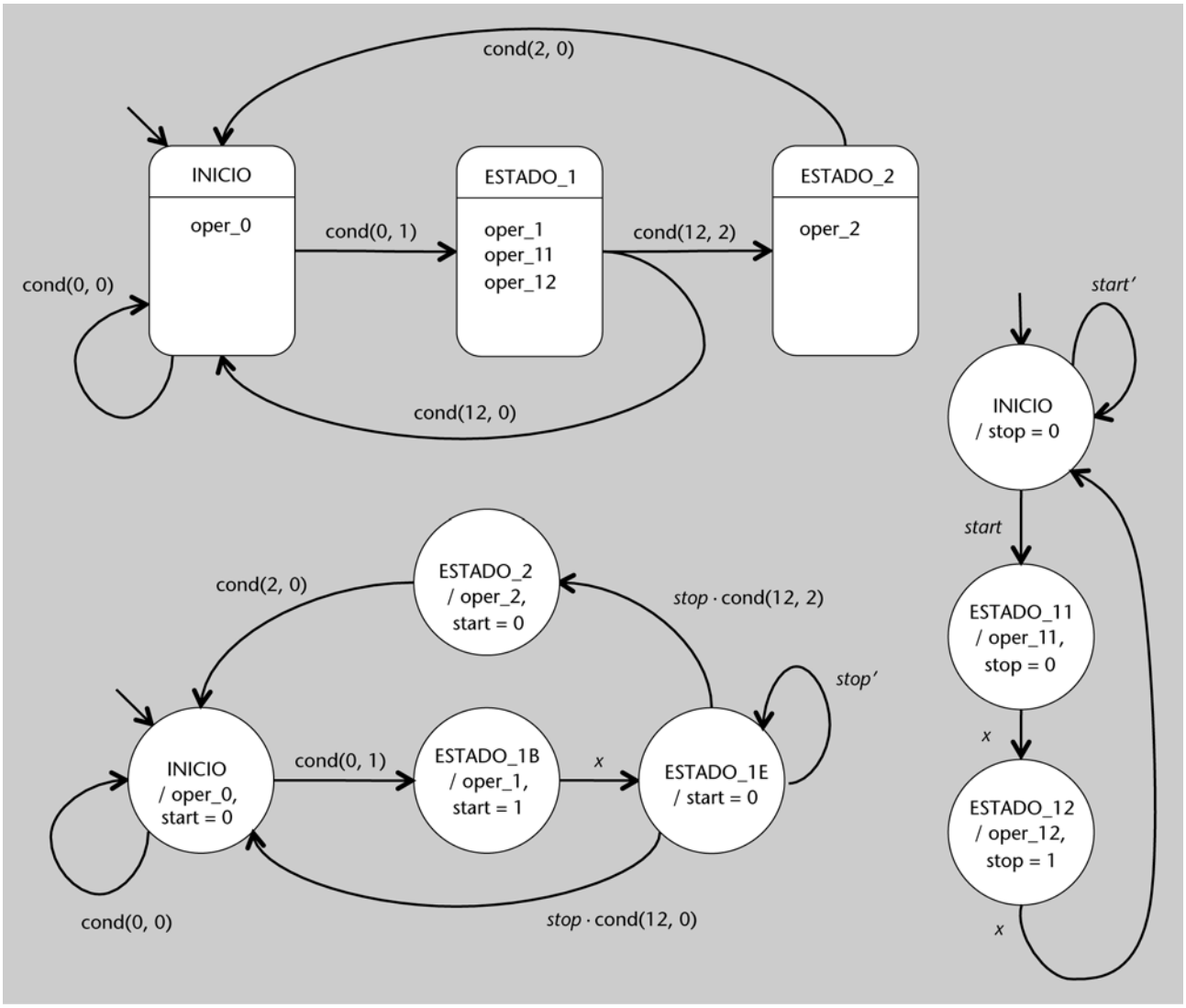
Como se ha visto, las relaciones entre las EFSM se establecen por medio de estados de espera y de referencias de estados externos a las entradas. Por ejemplo, que la EFSM de la parte inferior de la figura 23 pase de $MOVER_0$ a MOVER depende de una entrada que indique si la EFSM principal está o no en 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*.

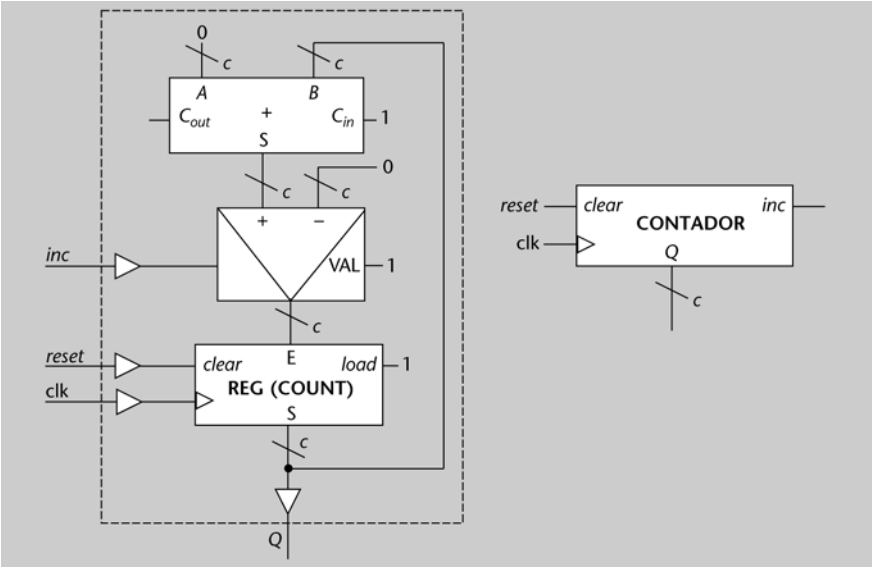
La materialización de estos circuitos es relativamente sencilla, ya que es suficiente con construir cada uno de las EFSM. Hay que recordar que la arquitectura de cada EFSM es de FSMD, con la unidad de control diferenciada de la unidad de proceso.

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.

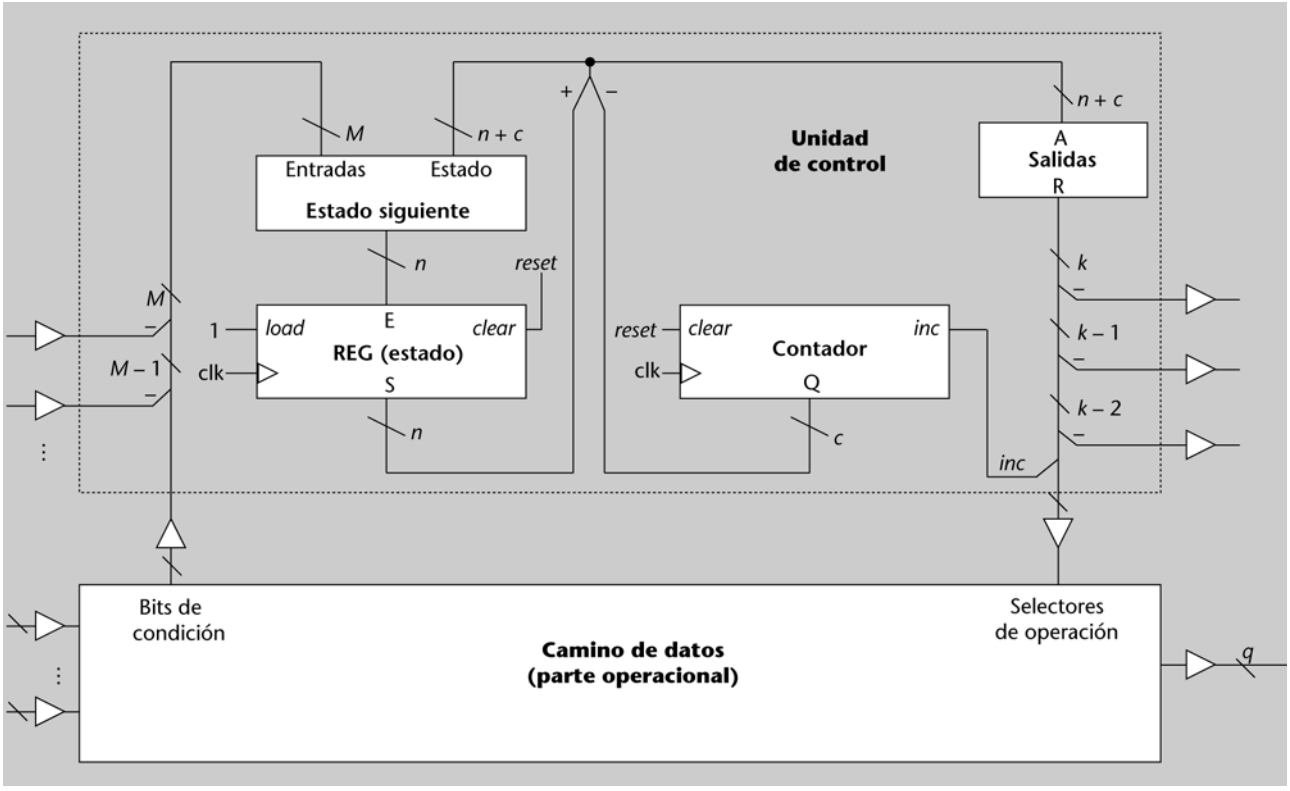
Así pues, la unidad de control está constituida por una máquina de estados compuesta de una FSM principal y distintas secundarias que comparten un único registro de estado, que es el registro interno del contador, COUNT. Eso es posible porque no están nunca activas al mismo tiempo. Consiguientemente, 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.

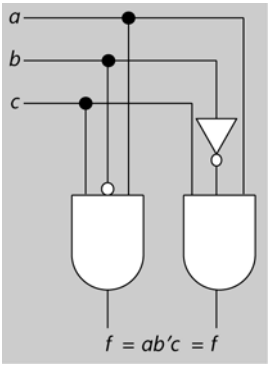
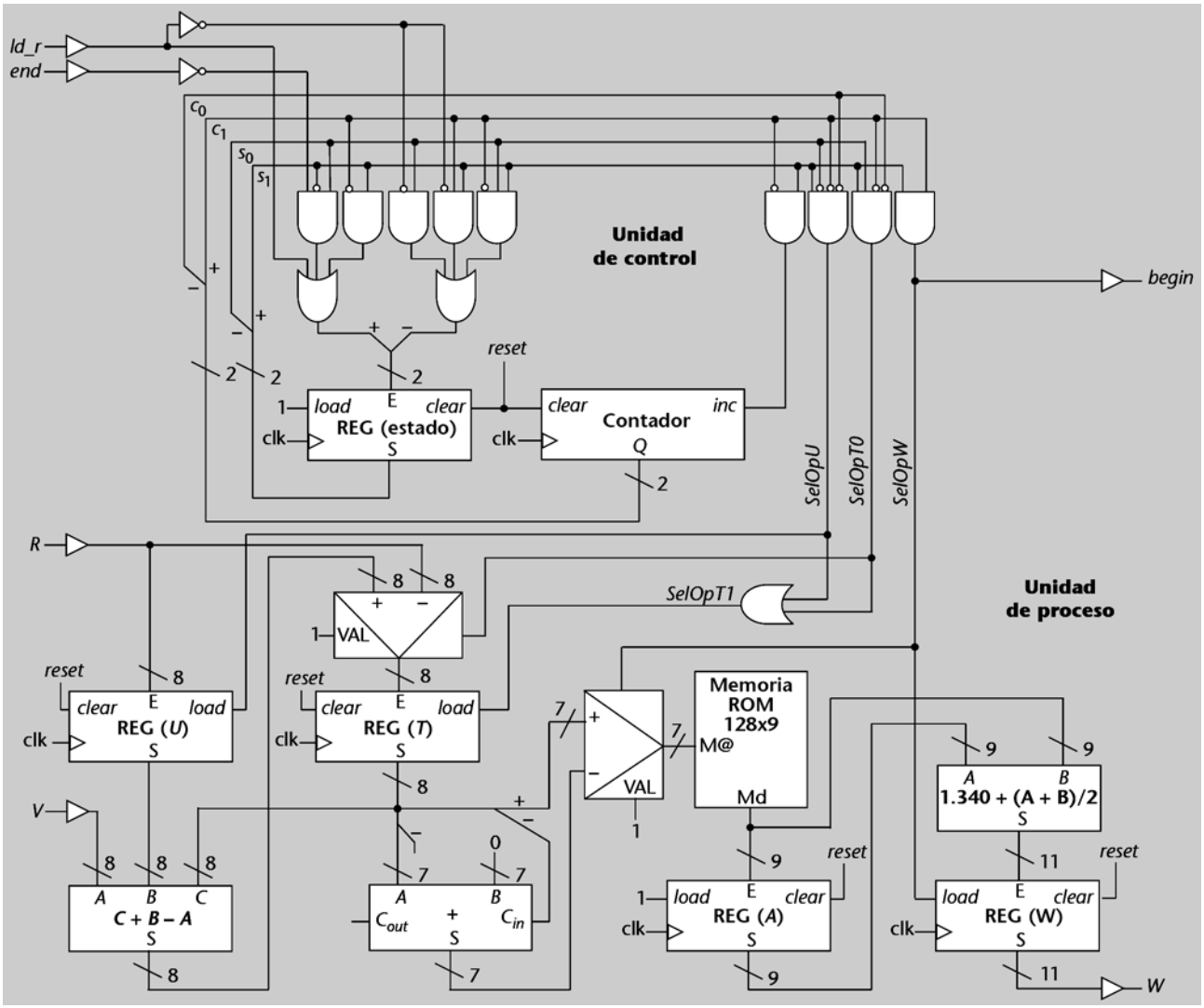


Figura 27. Circuito del control adaptativo de velocidad



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.

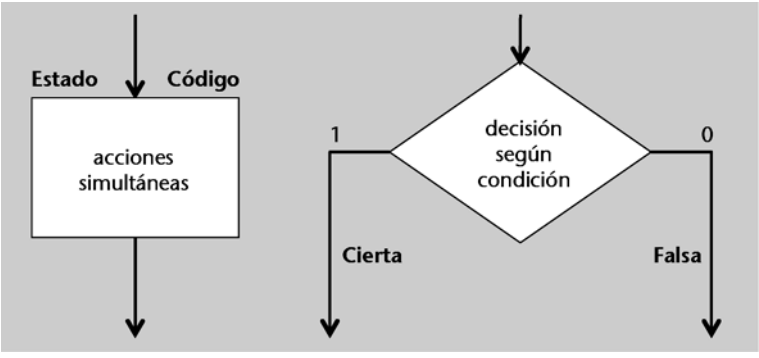
Visto desde otro punto de vista, los diagramas de estados tienen que ir acompañados, necesariamente, de las condiciones (entradas) y de las acciones (salidas) que se establecen para cada arco y nodo que tienen. Eso provoca que, cuando el número de entradas y salidas crece, resulten pesados de tratar. 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

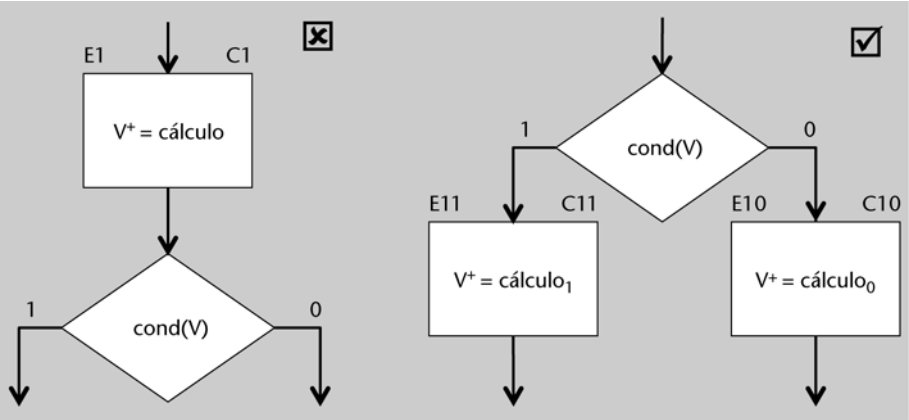


En comparación con los grafos que representan EFSM, las cajas de estados se corresponden a los nodos y las de decisión, a los arcos.

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.

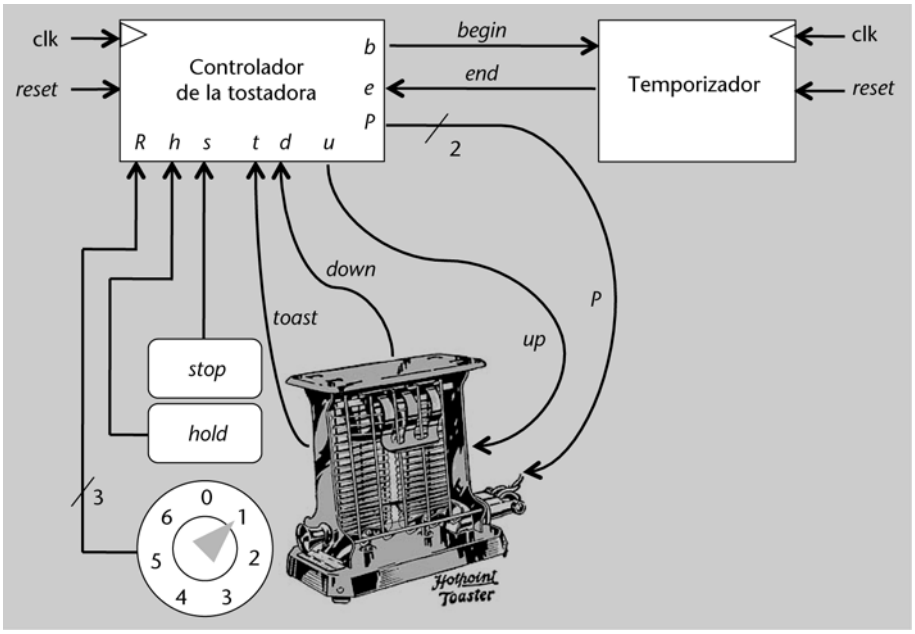
Figura 29. Formas desaconsejada y aconsejada de colocar estados con cálculos que afectan a bits de condición



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.

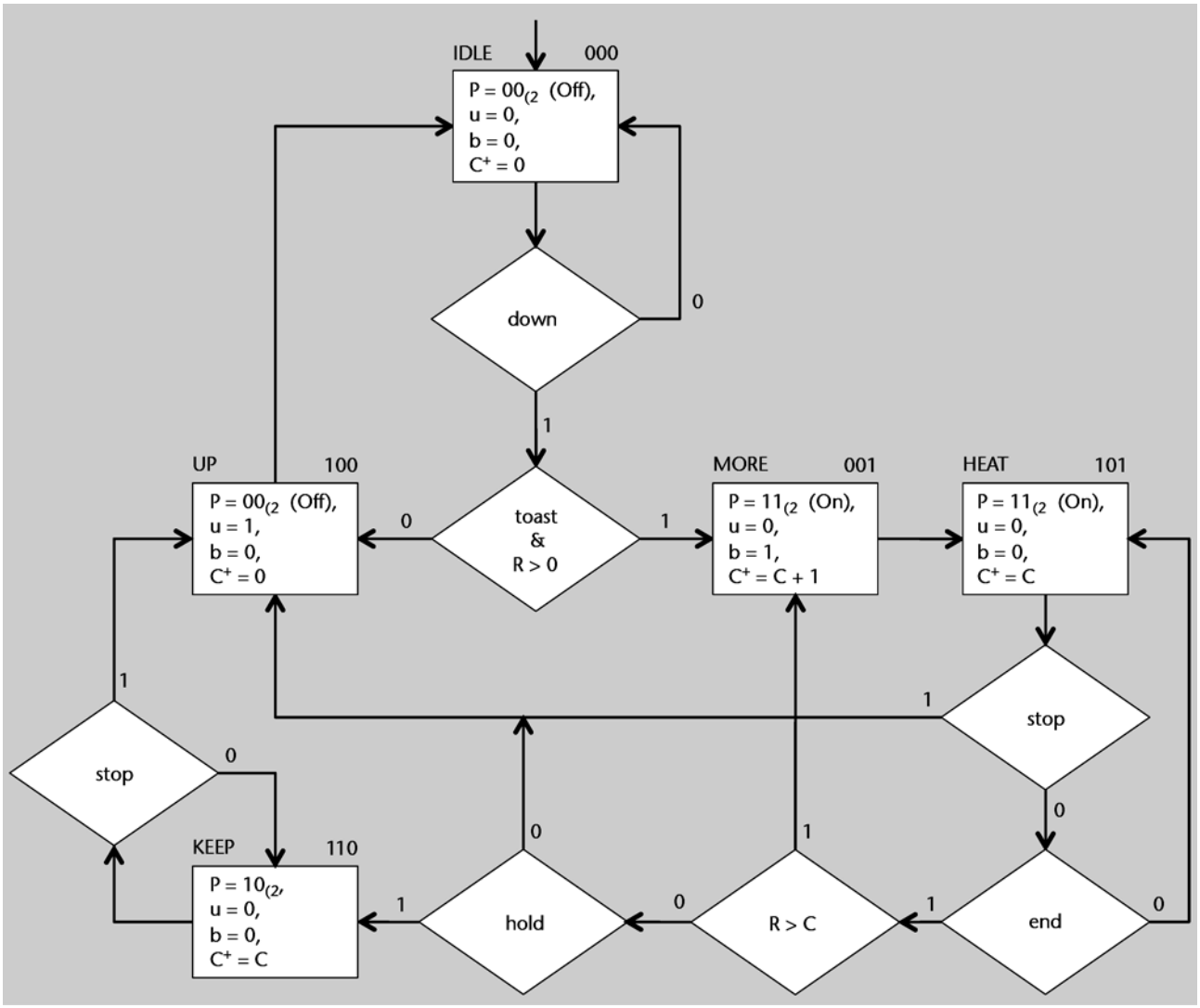
Es recomendable que las acciones y las condiciones que se describen en las cajas del diagrama de estados sean lo más generales posible. De esta manera son más inteligibles y no quedan vinculadas a una implementación concreta.

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:

$$s^+_3 = s^*_3$$
$$s^+_2 = s^*_2 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b$$
$$s^+_1 = s^*_1 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b$$
$$s^+_0 = s^*_0 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b$$

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:

$$s^+_3 = s_3 \oplus c_3$$
$$s^+_2 = s_2 \oplus c_2$$
$$s^+_1 = s_1 \oplus c_1$$
$$s^+_0 = s_0 \oplus c_0$$

donde c_i hace referencia a la función de cambio de bit. Hay que tener en cuenta que:

$$s_i \oplus 0 = s_i$$
$$s_i \oplus 1 = s'_i$$

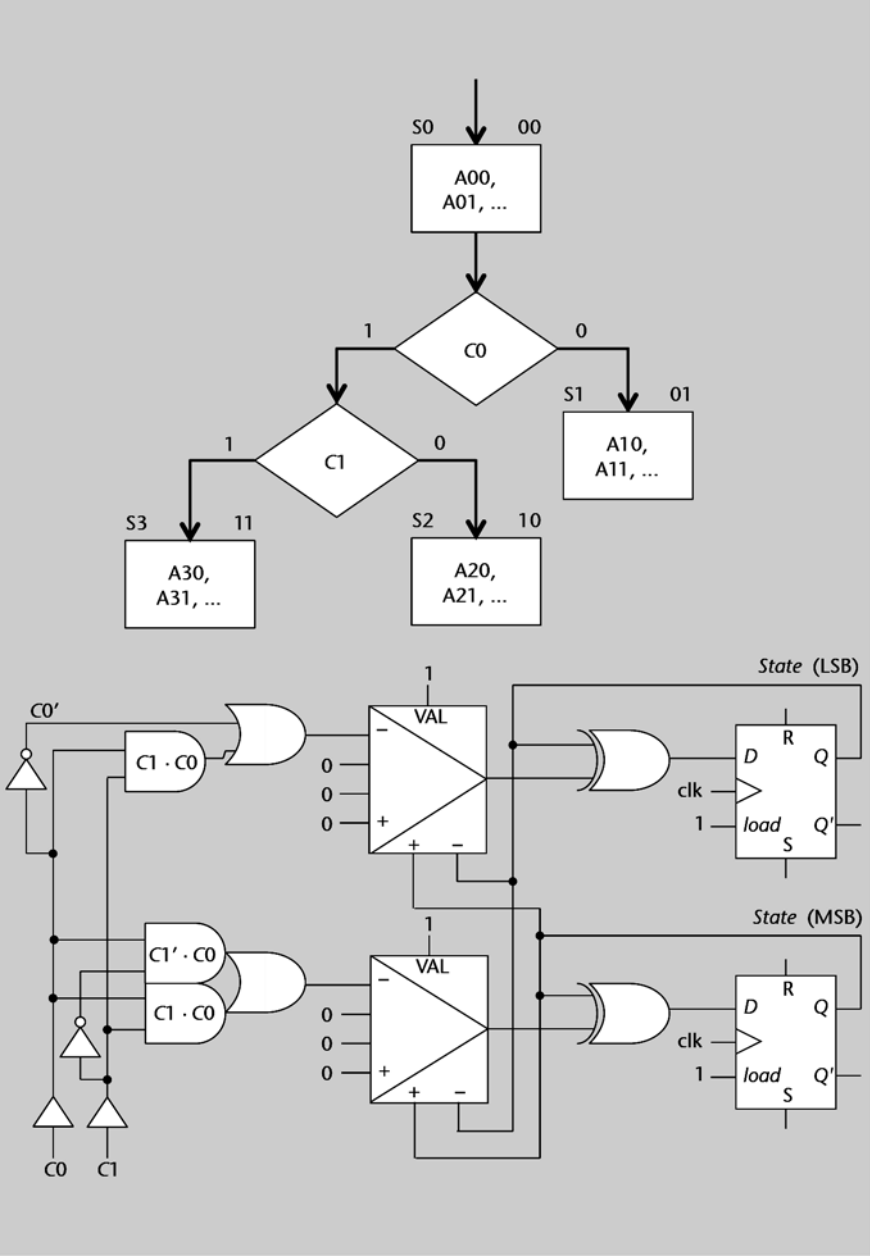
Siguiendo con el ejemplo, para construir las expresiones de las funciones de cambio c_i, se tiene que ver qué bits entre el código de estado origen y destino cambian y añadir el término correspondiente a la suma de productos que toque. En este caso, solo cambia el bit que hay en la posición 1:

$$c_3 = c^*_3$$
$$c_2 = c^*_2$$
$$c_1 = c^*_1 + s'_3 \cdot s_2 \cdot s'_1 \cdot s_0 \cdot a \cdot b$$
$$c_0 = c^*_0$$

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 ($[s'_2 s'_1 s'_0] \cdot dtr$) 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 $[s'_2 s'_1 s'_0] \cdot d(t' + r')$ 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.

$$s^+_2 = s_2 \oplus ([s'_2 s'_1 s'_0] \cdot d(t' + r') + [s'_2 s'_1 s_0] \cdot 1 + [s_2 s'_1 s'_0] \cdot 1 + [s_2 s'_1 s_0] \cdot s'er)$$
$$s^+_1 = s_1 \oplus ([s_2 s'_1 s_0] \cdot s'ehr' + [s_2 s_1 s'_0] \cdot s)$$
$$s^+_0 = s_0 \oplus ([s'_2 s'_1 s'_0] \cdot dtr + [s_2 s'_1 s_0] \cdot (s + er'))$$

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:

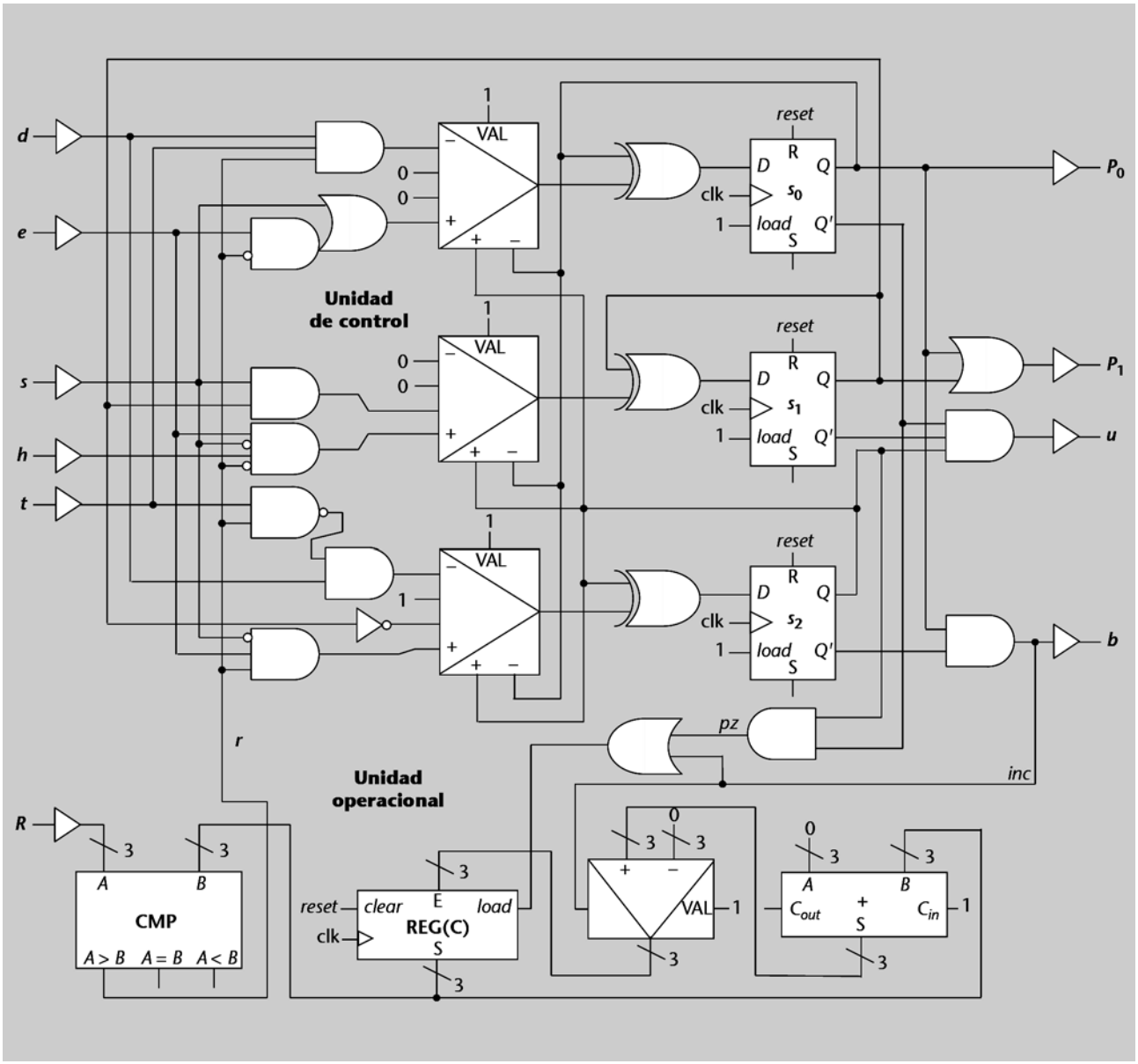
- Para un registro de cuenta C, $C^+ = 0$ y $C^+ = C + 1$, que se seleccionarán con las señales *inc*, de incremento, y *pz*, de poner a cero.
- La comparación con la referencia R, $R > C$, que da el bit de condición r.

Las funciones de salida (*inc*, *pz*, *u*, *b*, *P*) se calculan a partir del estado y son las siguientes:

$$inc = s'_2\, s'_1 s_0 \ \{= \text{MORE}\}$$
$$pz = s'_2\, s'_1 s'_0 + s_2 s'_1 s'_0 \ \{= \text{IDLE} + \text{UP}\}$$
$$u = s_2 s'_1 s'_0 \ \{= \text{UP}\}$$
$$b = s'_2\, s'_1 s_0 \ \{= \text{MORE}\}$$
$$p_1 = s'_2\, s'_1 s_0 + s_2 s'_1 s_0 + s_2 s_1 s'_0 \ \{= \text{MORE} + \text{HEAT} + \text{KEEP}\}$$
$$p_0 = s'_2\, s'_1 s_0 + s_2 s'_1 s_0 \ \{= \text{MORE} + \text{HEAT}\}$$

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.

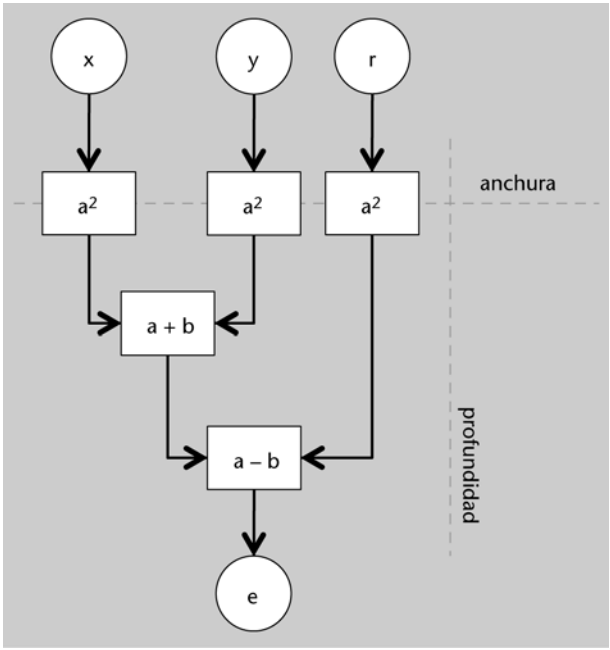
Los esquemas de cálculo son, de hecho, representaciones gráficas de cálculos más o menos complejos que se manipulan de cara a obtener una que sea adecuada para las condiciones del problema en el que se tengan que hacer. Unas veces interesará que el cálculo se haga muy rápido y otras, que se realice con 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

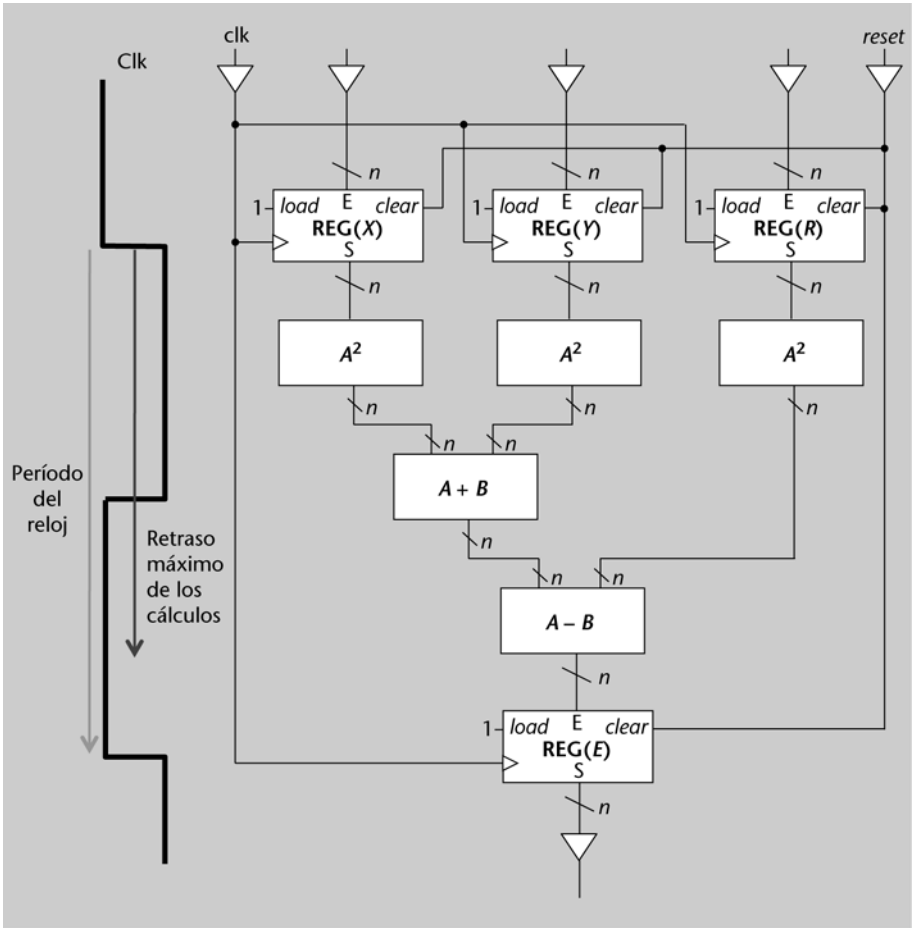
que los operandos se almacenan en registros, igual que el resultado, el cálculo se ve como una transferencia entre registros. Para el ejemplo anterior, sería:

$$E^{+} = X^2 + Y^2 - R^2$$

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.

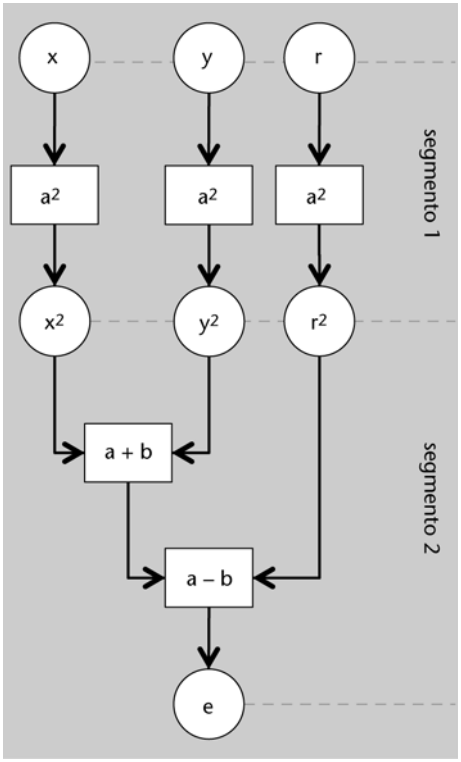
En la figura 36 aparece una segmentación del esquema de cálculo que se ha tomado como ejemplo en el apartado anterior. En el primer segmento se calculan los cuadrados de todos los datos y en el segundo se hacen las operaciones de suma y de resta. Para efectuar este cálculo se necesitan dos ciclos de 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.

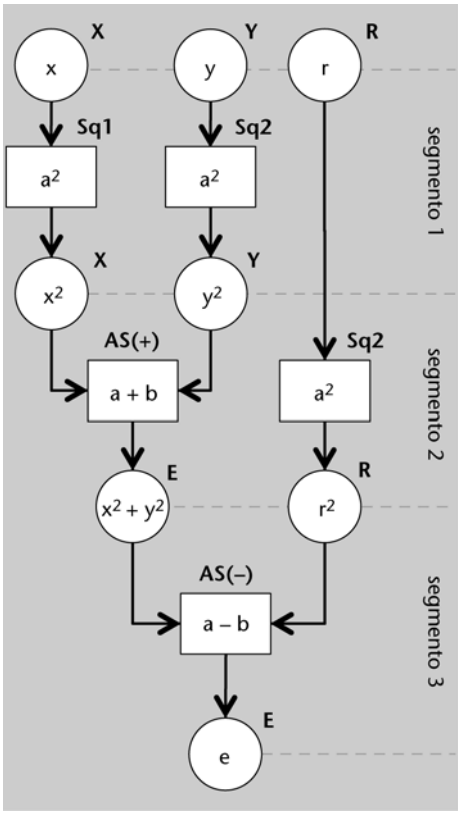
2.3. Esquemas de cálculo con recursos compartidos

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

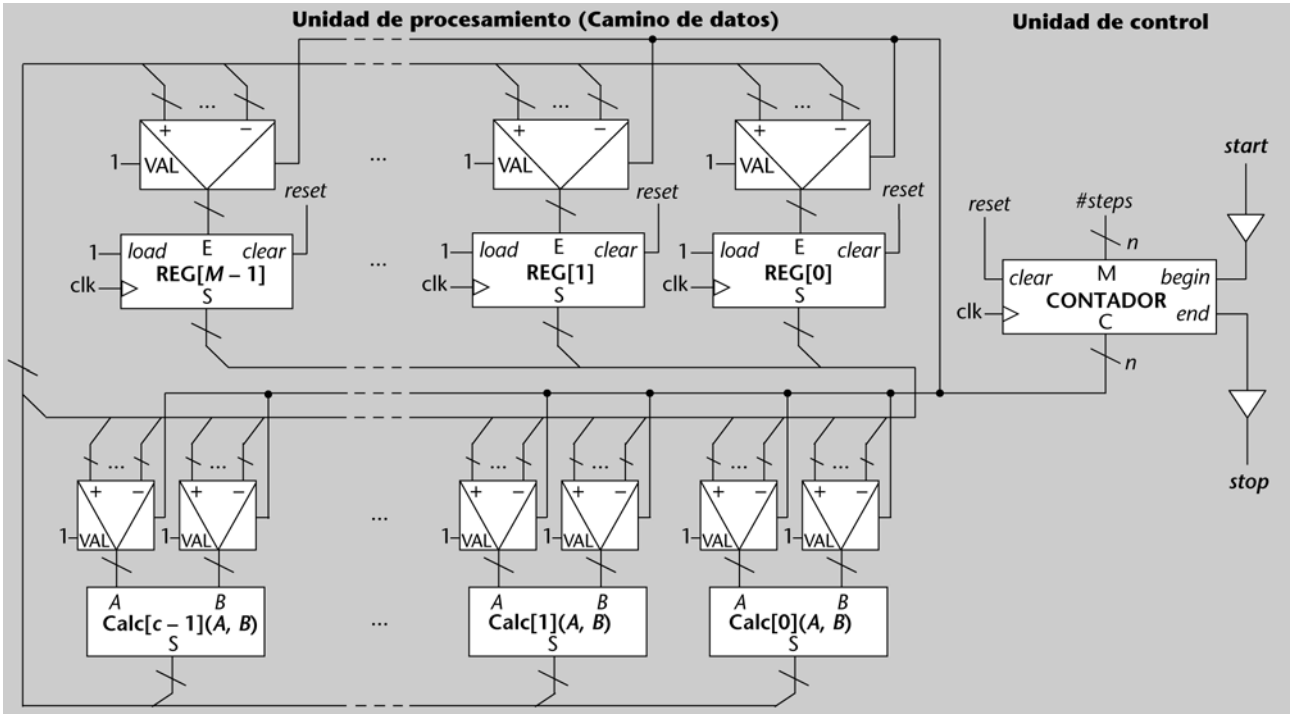
La materialización de un esquema de cálculo consiste en transformar su grafo etiquetado en el circuito correspondiente. Los circuitos se organizan de manera similar a como se ha visto para las máquinas de estados: se dividen en una unidad de procesamiento, que es el esquema de cálculo propiamente dicho, 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

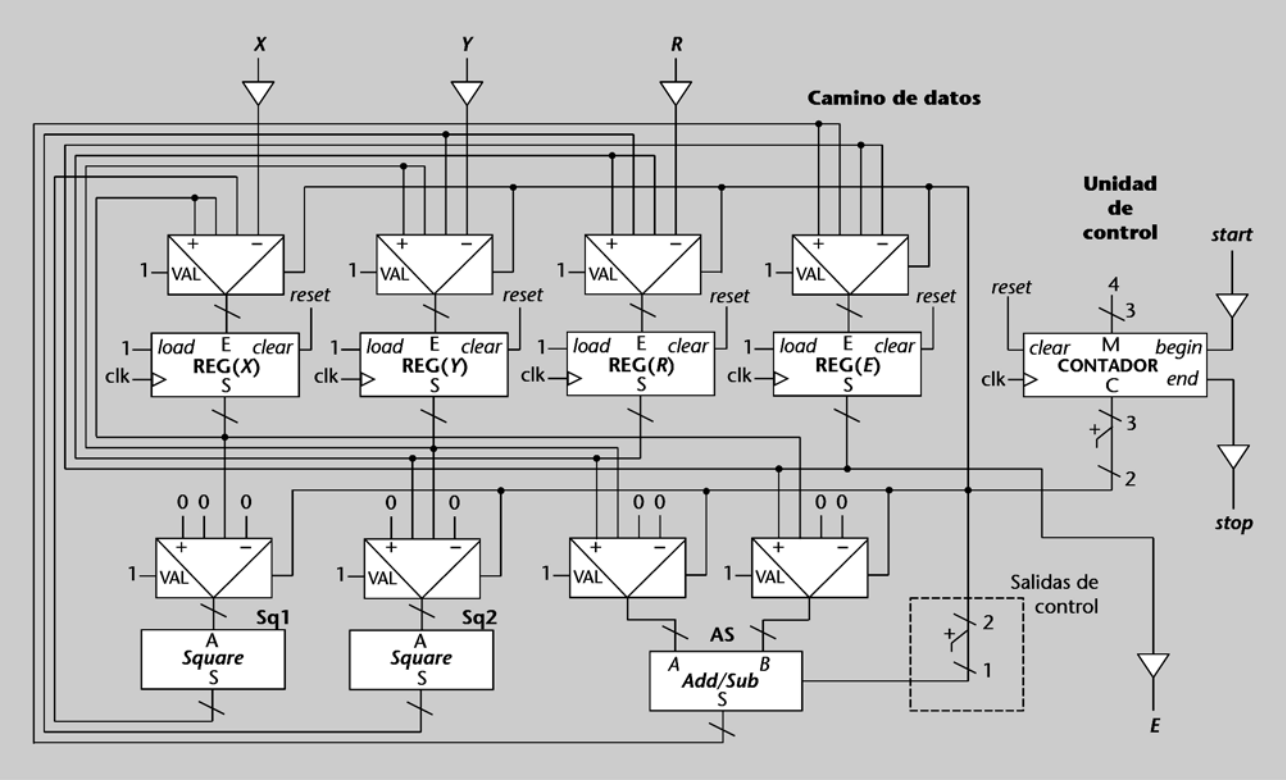


Hay dos buses de interconexión entre los recursos: el que recoge los datos de los recursos de cálculo y los lleva a los de memoria y el que sirve para la comunicación en sentido inverso. En la figura solo se ha marcado que cada elemento toma algunos bits del bus o los pone, sin especificar cuáles.

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:

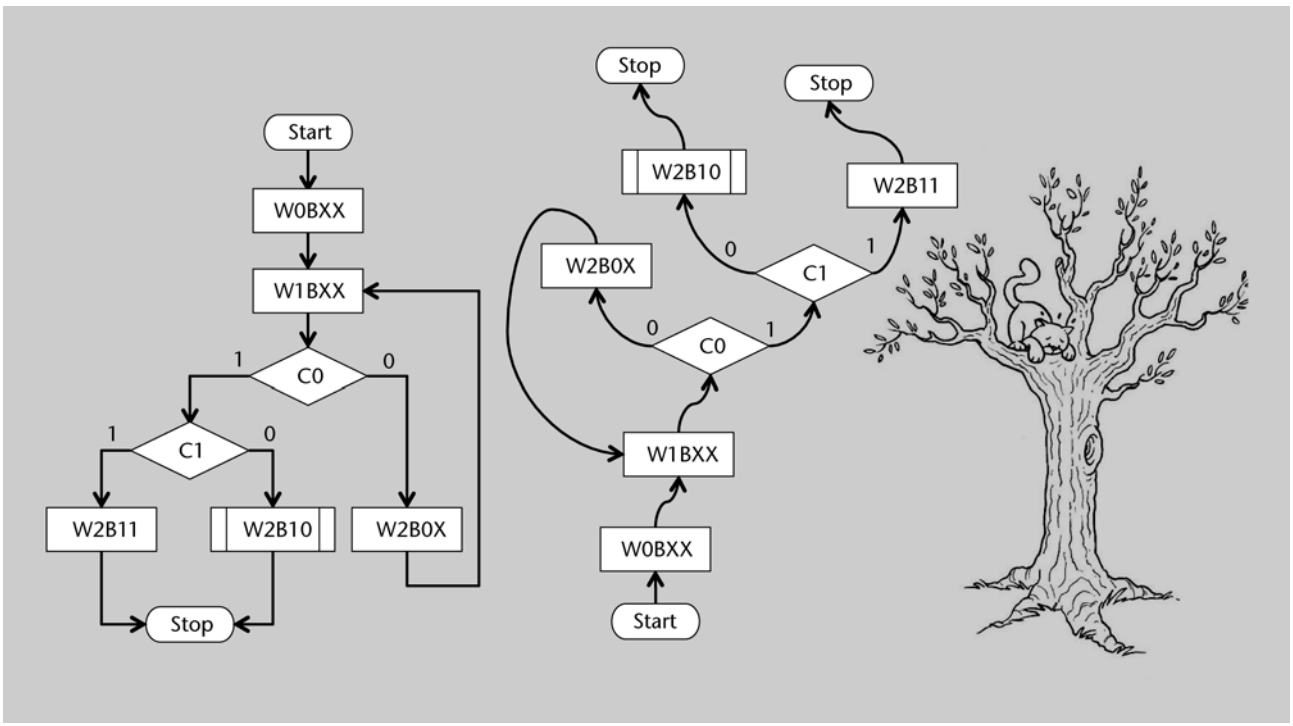
uno terminal, uno de procesamiento, uno de proceso predefinido (se explica qué es a continuación) y uno de decisión.

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.

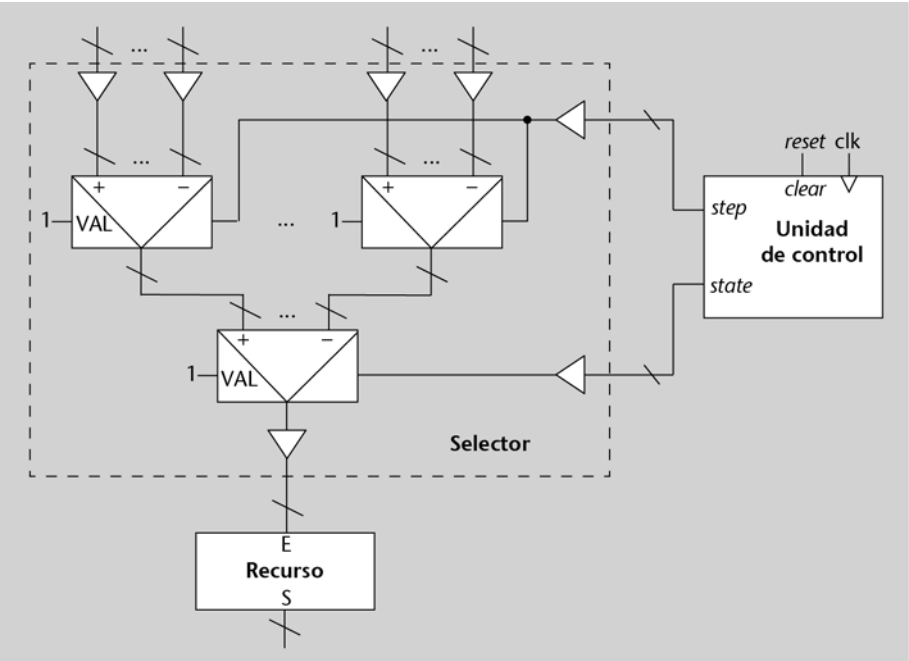
2.6. Materialización de máquinas algorítmicas

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, $s0$ y $s1$, 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 $C0$.

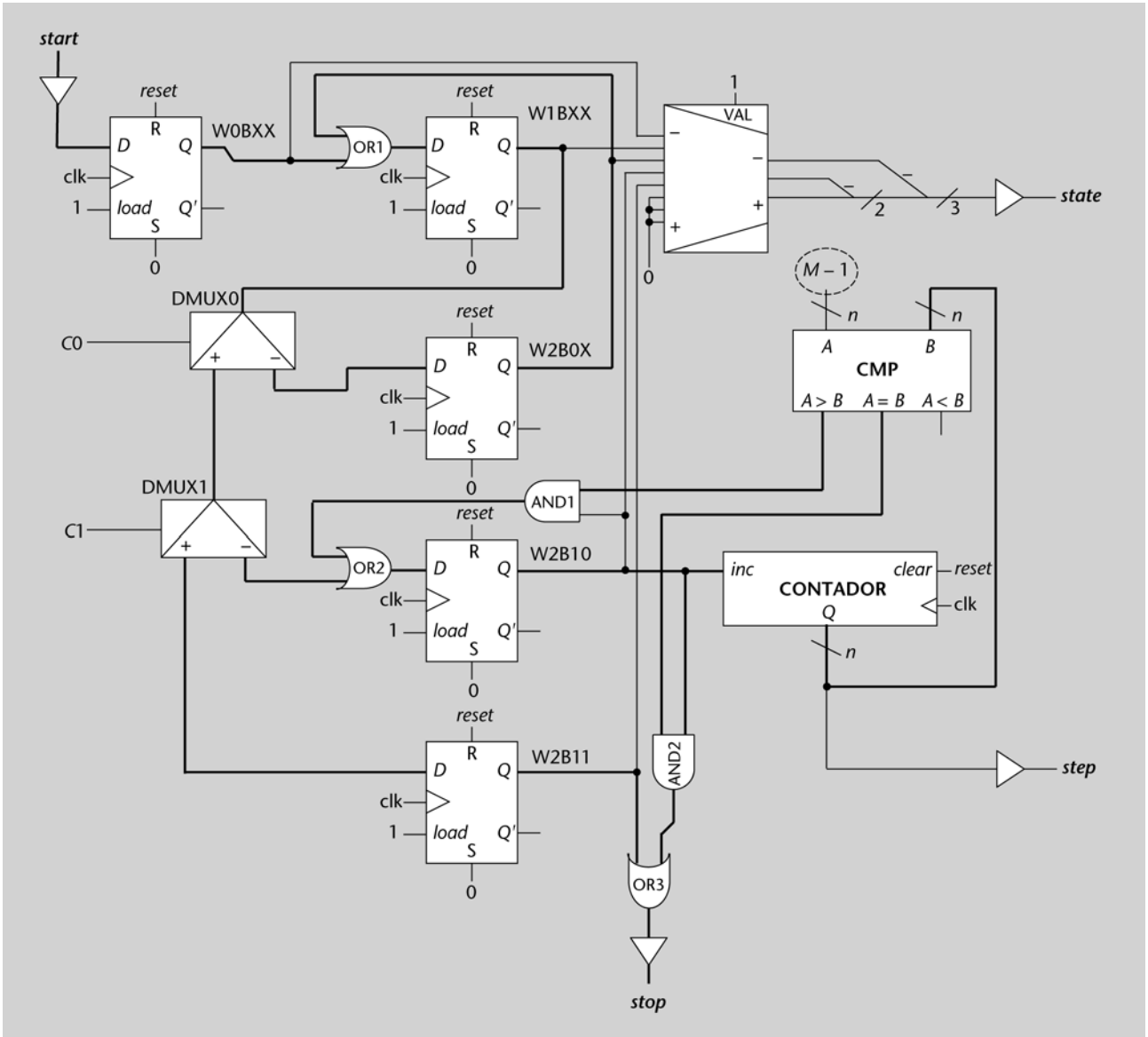
Los nodos que se corresponden con un proceso predefinido (con un esquema de cálculo, por ejemplo) activan un contador. Este contador es compartido por todos los nodos de proceso predefinido, ya que solo puede haber uno activo al mismo tiempo. En el ejemplo de la figura 40 vemos uno, que es W2B10.

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, $C0$ y $C1$. 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.

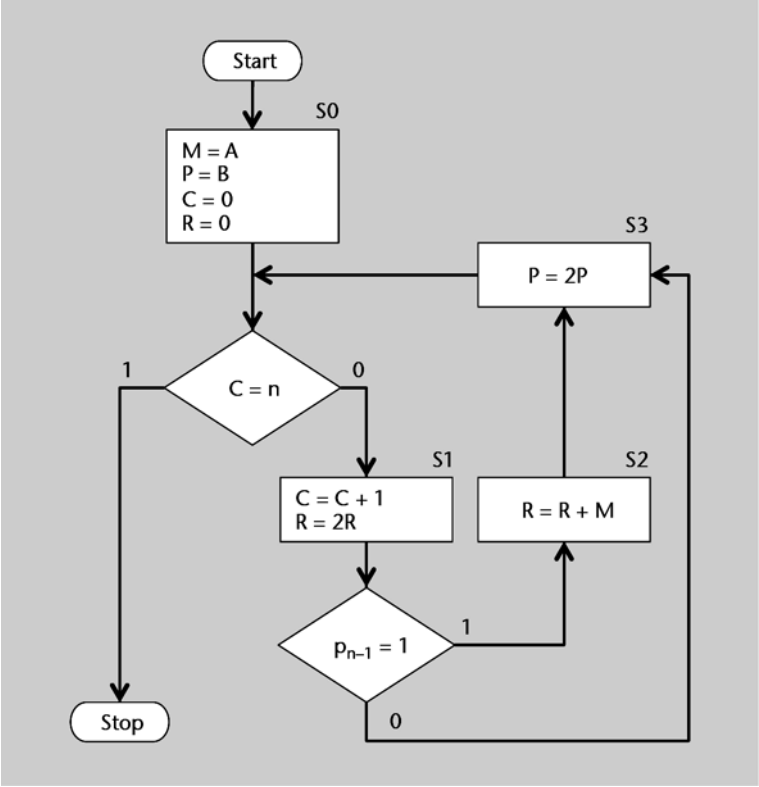
Para entenderlo, a continuación se realiza el producto de 1011 (multiplicando) por 1010 (multiplicador):

				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.)

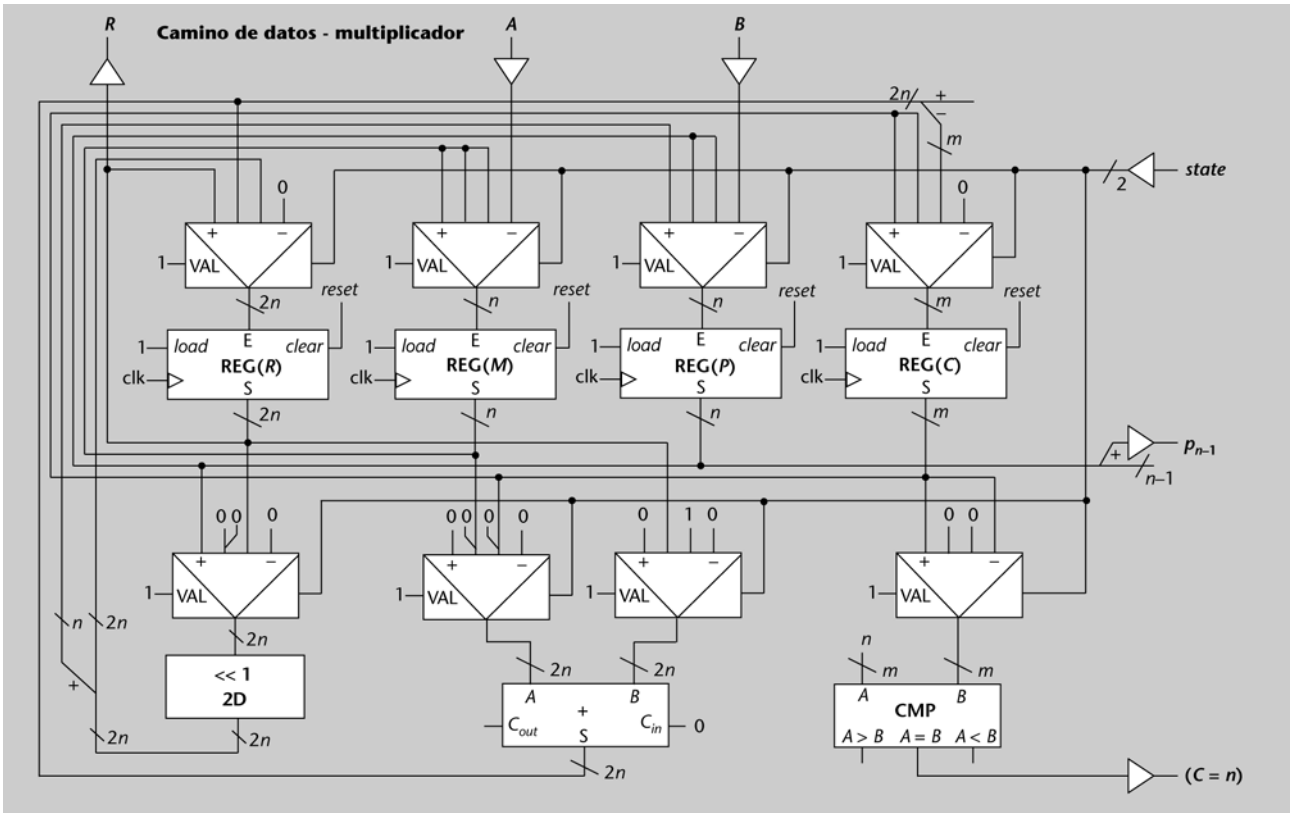
				1	0	1	1	M
			x	1	0	1	0	P
0	0	0	0	0	0	0	0	$R = 2 \cdot R$
			+	1	0	1	1	$M \cdot p_{n-1}$
0	0	0	0	1	0	1	1	$R = R + M \cdot p_{n-1}$
0	0	0	1	0	1	1	0	$R = 2 \cdot R$
			+	0	0	0	0	$M \cdot p_{n-2}$
0	0	0	1	0	1	1	0	$R = R + M \cdot p_{n-2}$
0	0	1	0	1	1	0	0	$R = 2 \cdot R$
			+	1	0	1	1	$M \cdot p_{n-3}$
0	0	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

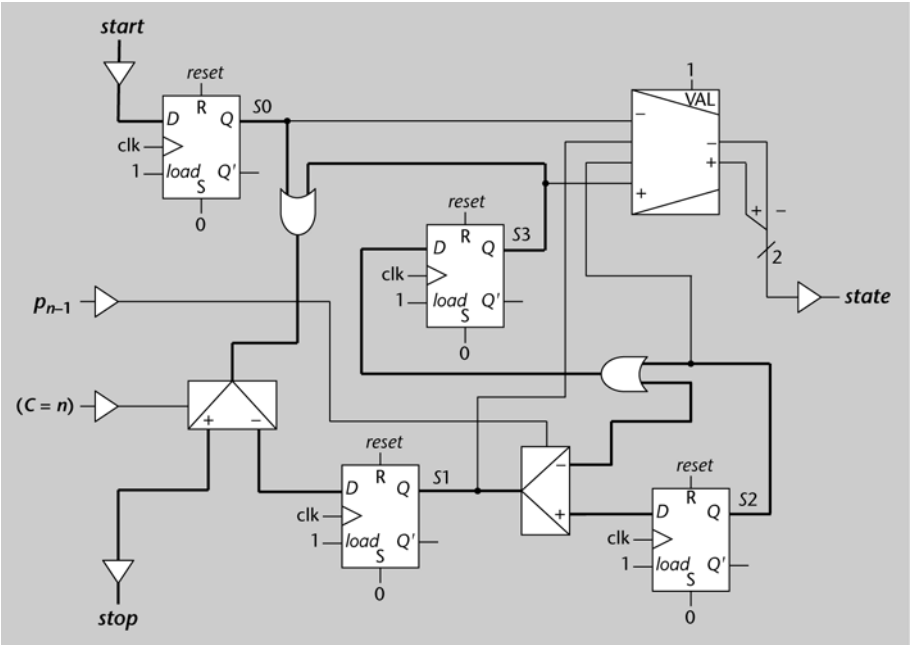


La unidad de procesamiento mostrada en la figura 44 sigue fielmente el modelo de construcción, pero se puede simplificar con respecto al uso de los multiplexores.

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.

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.

> 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.

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.

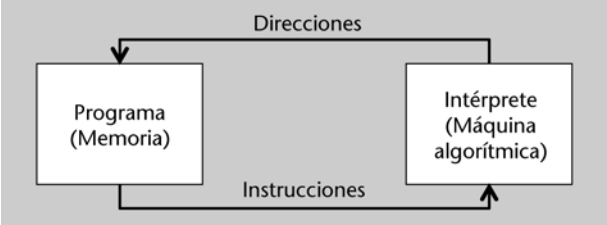
Así pues, es conveniente utilizar máquinas más genéricas, que puedan ejecutar conjuntos de funciones diferentes según la aplicación a la que se destinen o la propia evolución del producto en el que se encuentren. Para llevarlo a cabo, los algoritmos deberían estar almacenados en módulos de memoria, de 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.

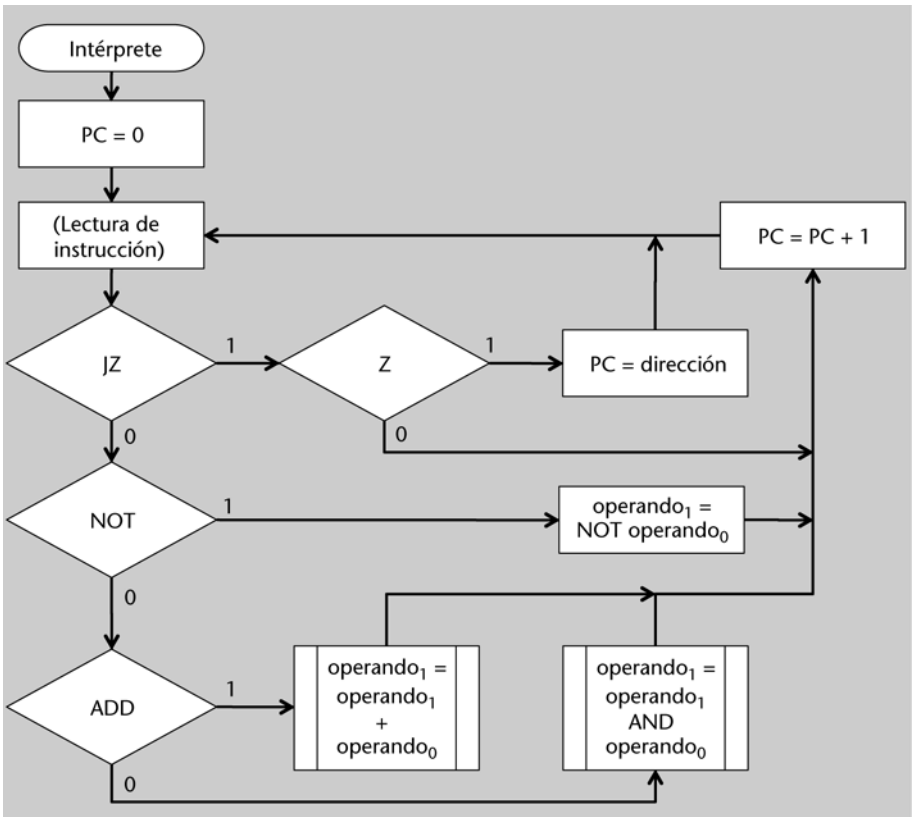
El resultado de las operaciones se almacenará en el $operando_1$. Si es *R*0 o *R*1, el resultado se perderá, pero el bit que identifica si la última operación ha sido cero o no (*Z*) sí que se actualizará. *Z* identifica un biestable que almacena esta condición.

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: $B = BR[Q_{2-0}]$; 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}
$operando_1$ = $operando_1$ AND $operando_0$	0: $B = BR[Q_{2-0}]$; 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 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, que, a su vez, depende de PC, que se modifica justo al estado anterior y, por lo tanto, no se actualiza hasta el comienzo del estado actual. 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 9 estados: "PC = 0", "PC = dirección", "PC = PC + 1", "lectura", "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 8.

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 7.

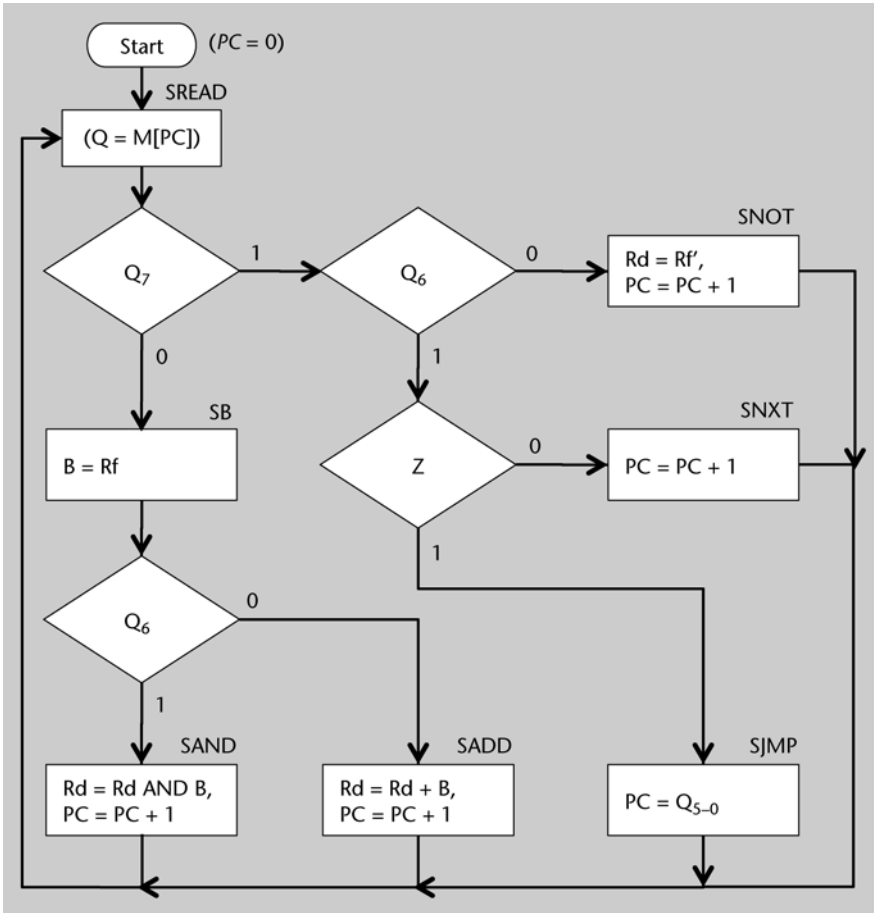
En la figura 48 se puede comprobar esta simplificación: se ha reducido el número de nodos y minimizado los esquemas de cálculo. En la figura se utiliza Rd (registro de destino) para representar el registro en posición $Q_{5–3}$ del banco de registros (BR), que sustituye el símbolo $operando_1$, y Rf (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-

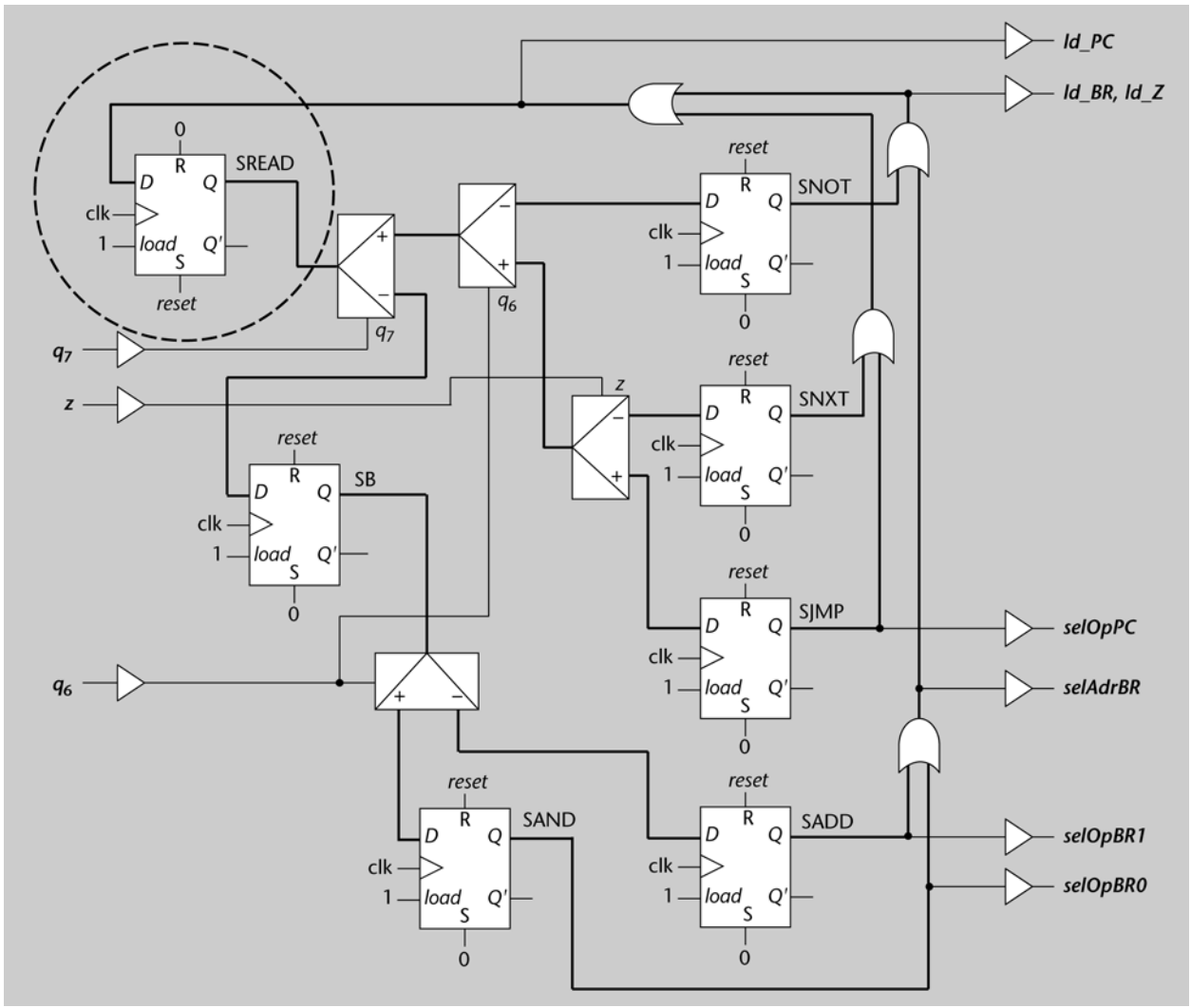
minan con "selOp" previo al nombre del recurso de memoria al que le proporcionan la entrada de datos. En el caso del banco de registros, *selOpBR* hace referencia a la selección del resultado que se debe escribir, *ld_BR*, a la señal que activa la escritura, y *selAdrBR* a la dirección del registro cuyo 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*
SREAD	0	0	x	xx	0	x	0
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 también se ocupa de proporcionar el primer 1, justo después de un *reset*, puesto que esta señal está conectada a la entrada set, de puesta asíncrona a 1, del biestable.

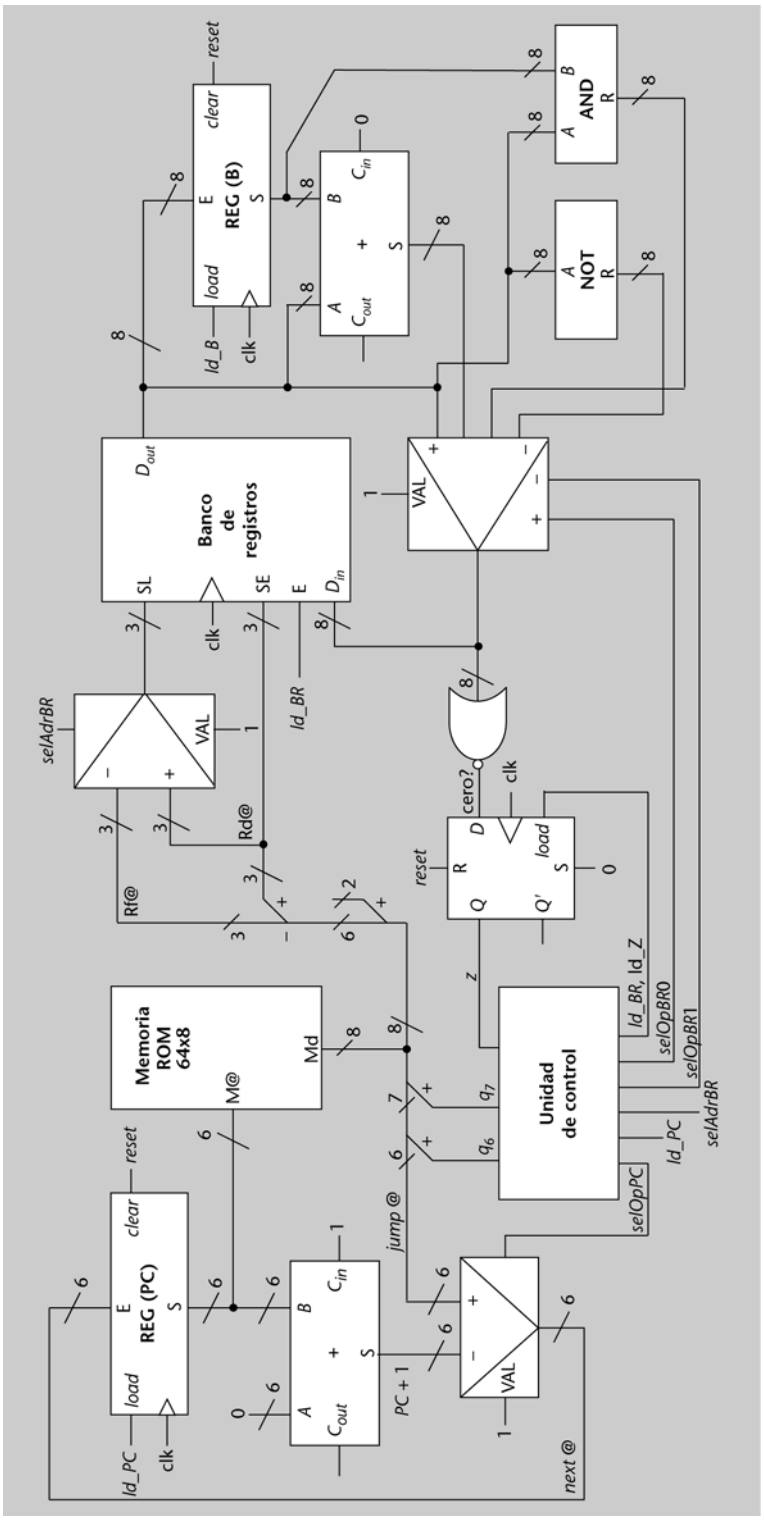
Figura 49. Unidad de control del Femtoproc, con codificación de estados de tipo *one-hot bit*



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 *R6* y *R7* se han sustituido por los valores de entrada del programa (es decir, son registros solo de lectura) y que el resultado queda en *R5*. Hay que tener presente que, inicialmente, todos los registros, excepto *R0*, *R1*, *R6* y *R7*, están a 0.

El programa calcula el máximo común divisor mediante restas sucesivas, según la expresión siguiente:

$$\text{MCD}(a, b) = \begin{cases} \text{MCD}(a - b, b), \text{ si } a > b \\ \text{MCD}(a, b - a), \text{ si } b > a \\ a, \text{ si } b = 0 \\ b, \text{ si } a = 0 \end{cases}$$

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	–	*R0* = 1000 0000 = 80h, bit de signo
---	configuración	–	*R1* = 0000 0001 = 01h, bit unitario
---	configuración	–	*R5*, registro de salida
---	configuración	–	*R6*, registro de entrada
---	configuración	–	*R7*, 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	*R4* = NOT(*R2*)
05h	ADD R4, R1	00 100 001	*R4* = –*R2* = Ca2(*R2*) = NOT(*R2*) + 1
06h	ADD R4, R3	00 100 011	*R4* = *R3* – *R2*
07h	AND R0, R4	01 000 100	Comprueba el bit de signo, el resultado no se guarda, porque *R0* es solo de lectura
08h	JZ 0Dh (pos)	11 001101	Si es positivo (si *R3* > *R2*) 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 *R2*: *R2* = –(*R3* – *R2*)
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		*R3* = *R4* = *R3* – *R2*
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`		*R5* = *R2* + *R3*, 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.)

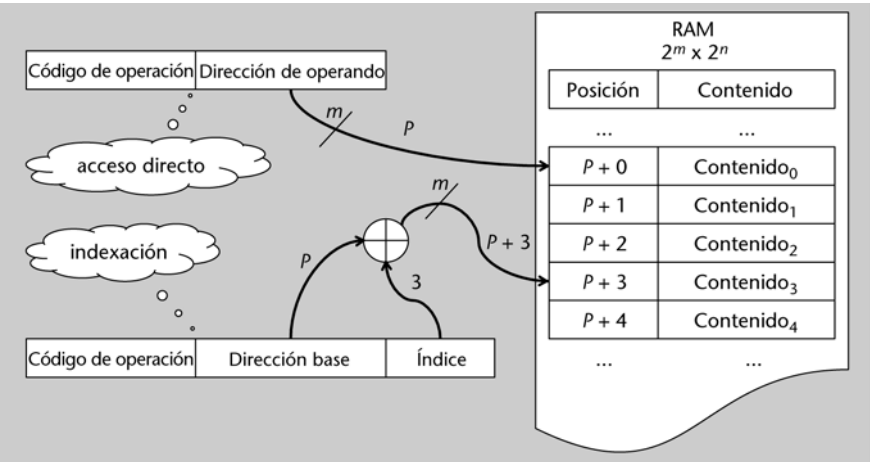
La memoria de programas ha de tener mucha más capacidad y, en consecuencia, las direcciones deben ser de más bits. Hay que tener en cuenta, en este sentido, que la codificación de las instrucciones también se realiza con más bits, ya que el repertorio se ha ampliado.

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.

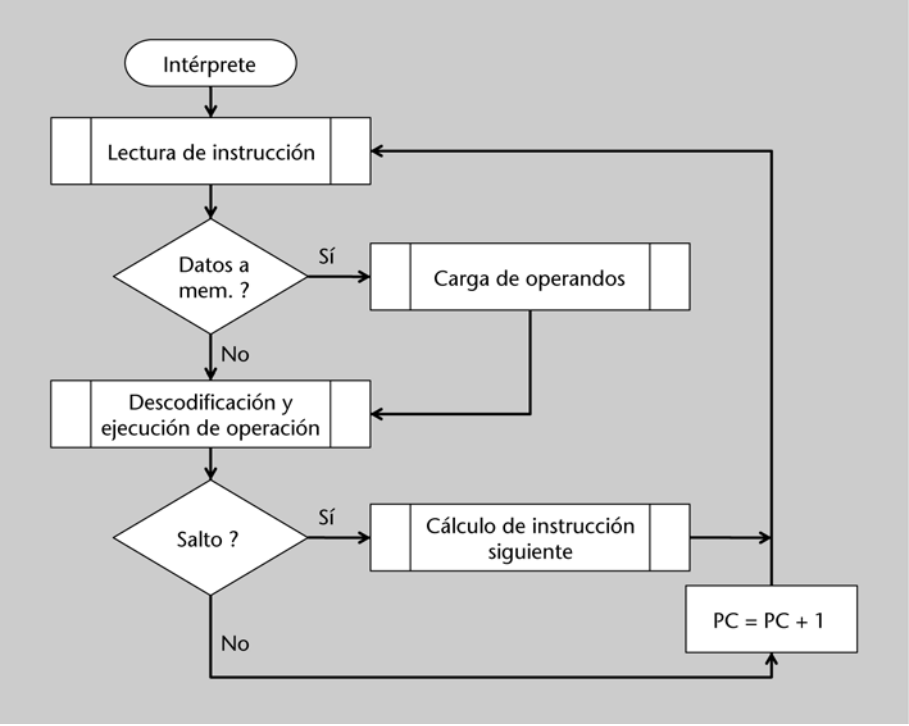
Hay que tener presente que la lectura de una sola instrucción puede necesitar varios pasos, dado que la anchura en bits debe permitir codificar un repertorio grande, lo que provoca que pueda ocupar más de una palabra de memoria.

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.

Microinstrucción	**Bits (1 por microorden)**															
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
EXEC	0	selectores de entrada, cargas de registros, operación de la ALU...														
JMPIF	1	selectores de condición				–	–	dirección de salto (microinstrucción siguiente)								

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.

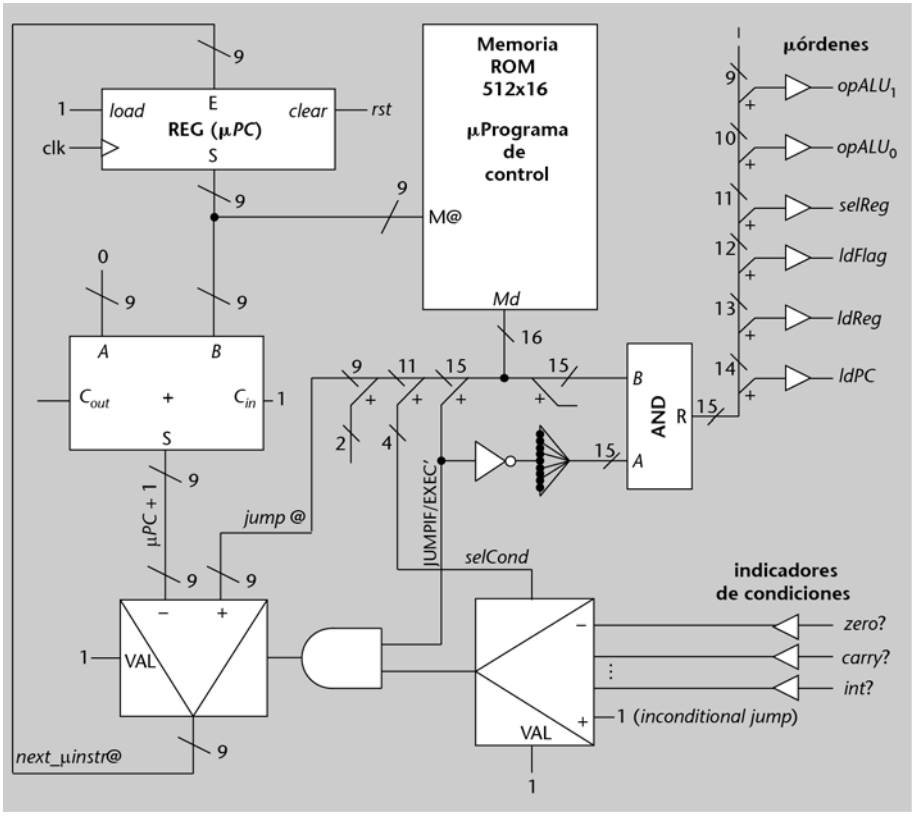
En la figura 53 aparece el esquema de un circuito secuenciador para el repertorio de microinstrucciones que se acaba de comentar. Se trata de un secuenciador paralelo, en el que cada microorden tiene un bit asociado. Por este

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_*μ*inst@*.

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-

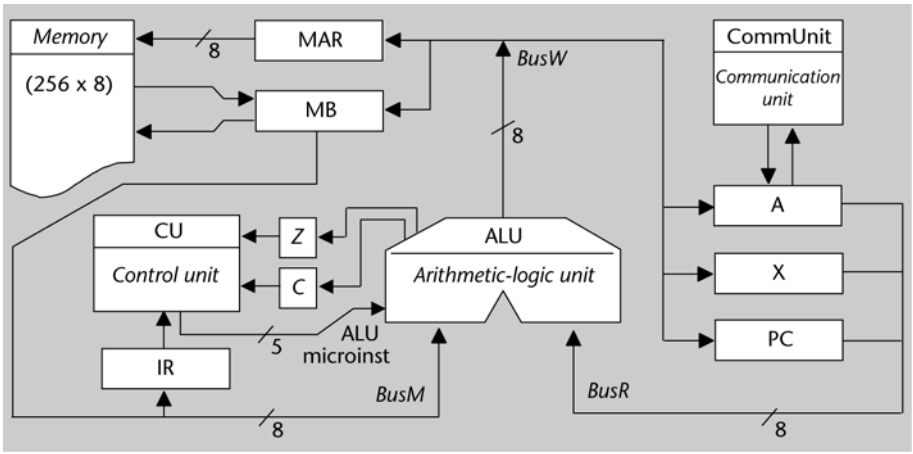
lación de bits y, finalmente, las de control de flujo. Como es habitual, las operaciones aritméticas se efectúan considerando que los números se representan en complemento a 2.

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

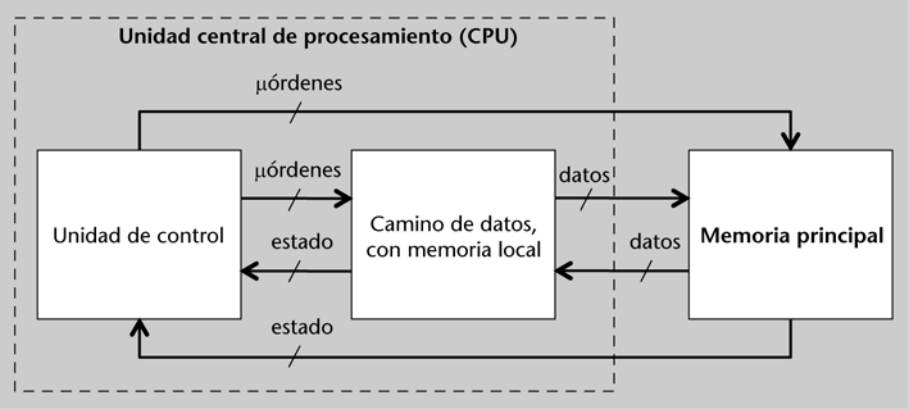
Dada la necesidad de disponer de procesadores con memorias de gran capacidad, se construyen de manera que la parte procesadora tenga recursos de memoria suficientes para trabajar de un modo eficiente (es decir, que el número de instrucciones ejecutadas por unidad de tiempo sea el mayor posible), pero 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.

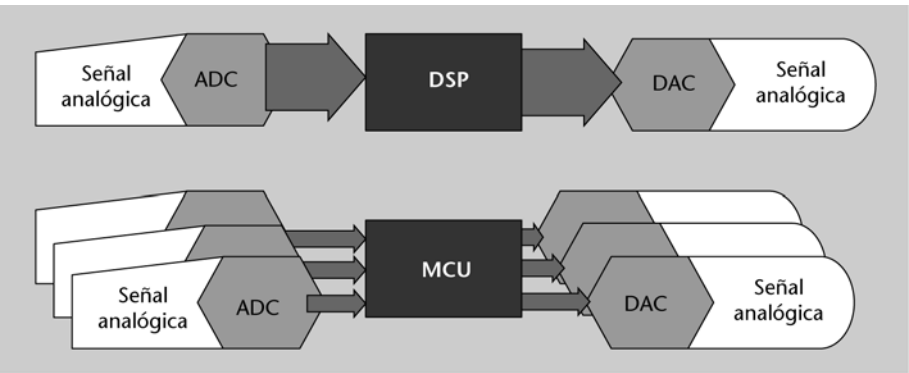
Para el caso particular de las imágenes, hay DSP específicos, denominados **unidades de procesamiento de imágenes** o **GPU** (del inglés, *graphics processing units*) que tienen una ISA similar, pero que suelen utilizar un sistema de memoria diferente con las direcciones organizadas en 2D para facilitar 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".

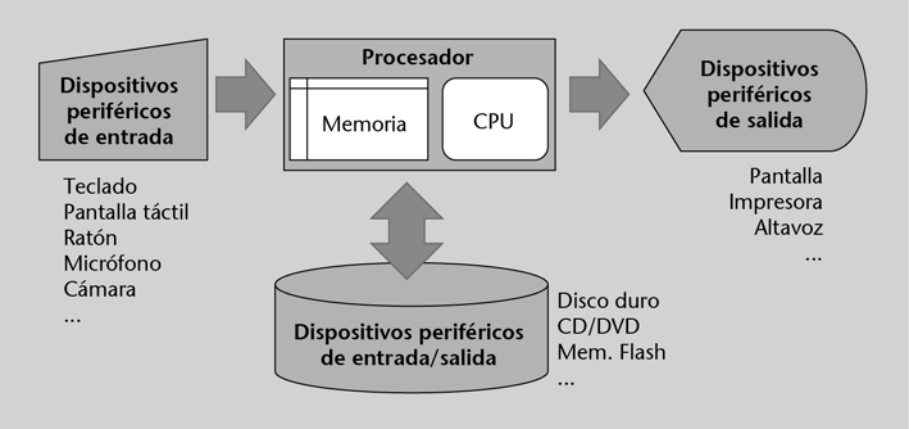
Sean como sean, los computadores tienen unos rasgos comunes que se comentarán en el resto de la sección.

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.

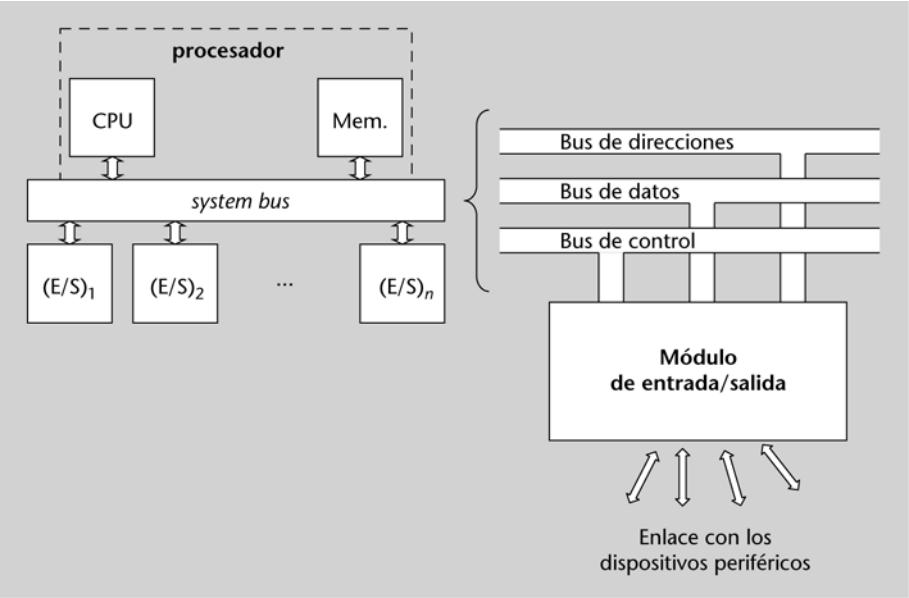
Para comunicarse con el procesador, los dispositivos periféricos disponen de controladores que, además de gobernar su funcionamiento, se relacionan con un **módulo de entrada/salida** del computador correspondiente. Estos módulos hacen de puente entre el periférico y el procesador, adaptando las diferentes 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.

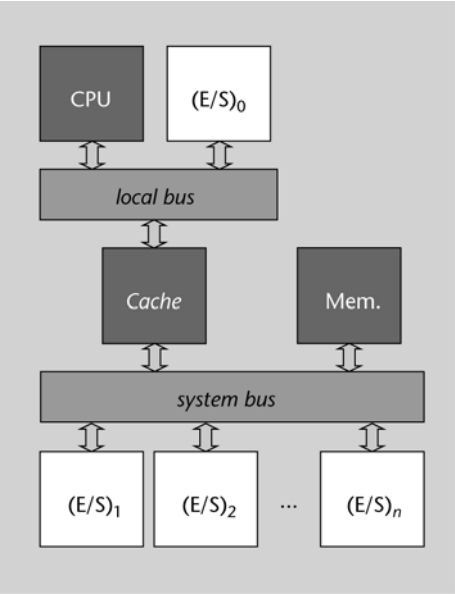
Las operaciones de entrada/salida de datos programadas implicarían sincronizar el procesador con el periférico correspondiente. Es decir, que el procesador debería esperar la disponibilidad del periférico para hacer la transmisión de los datos, lo que significaría una pérdida notable de rendimiento en 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

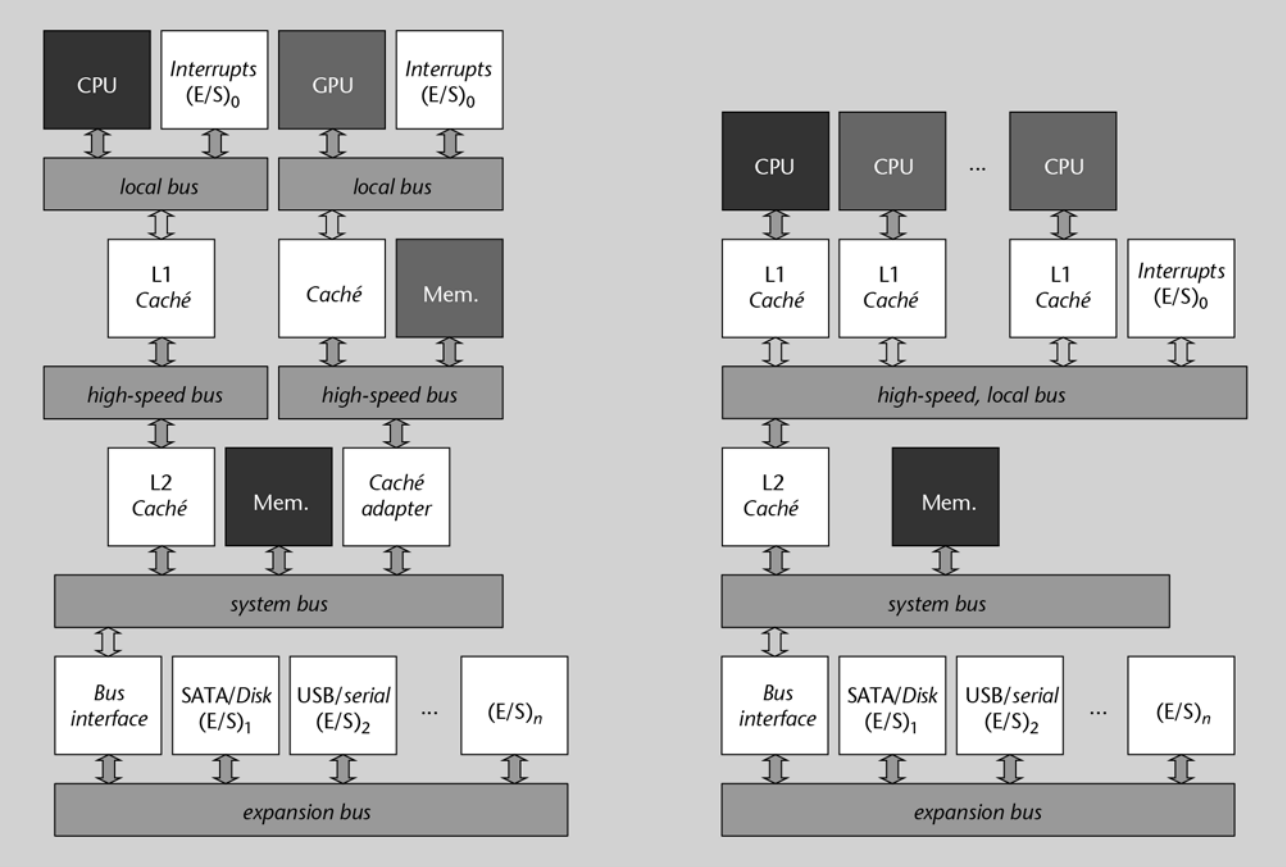
Un **computador de alto rendimiento** es aquel capaz de ejecutar en un mismo tiempo programas más grandes en memoria y número de instrucciones que los normales, entendiendo por normales los que son mayoría en un momento determinado.

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

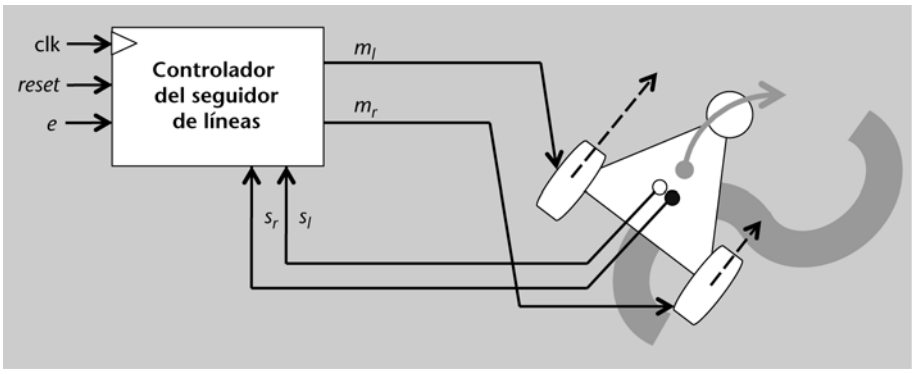
Los problemas que se proponen a continuación se deberían resolver una vez acabado el estudio del módulo, de manera que sirvan para comprobar el grado de consecución de los objetivos indicados al principio. No obstante, los cuatro primeros se pueden aprovechar como preparación de una posible práctica 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
[0º, 45º)	0	0	0
[45º, 90º)	0	0	1
[90º, 135º)	0	1	0
[135º, 180º)	0	1	1
[180º, 225º)	1	0	0
[225º, 270º)	1	0	1
[270º, 315º)	1	1	0
[315º, 360º)	1	1	1

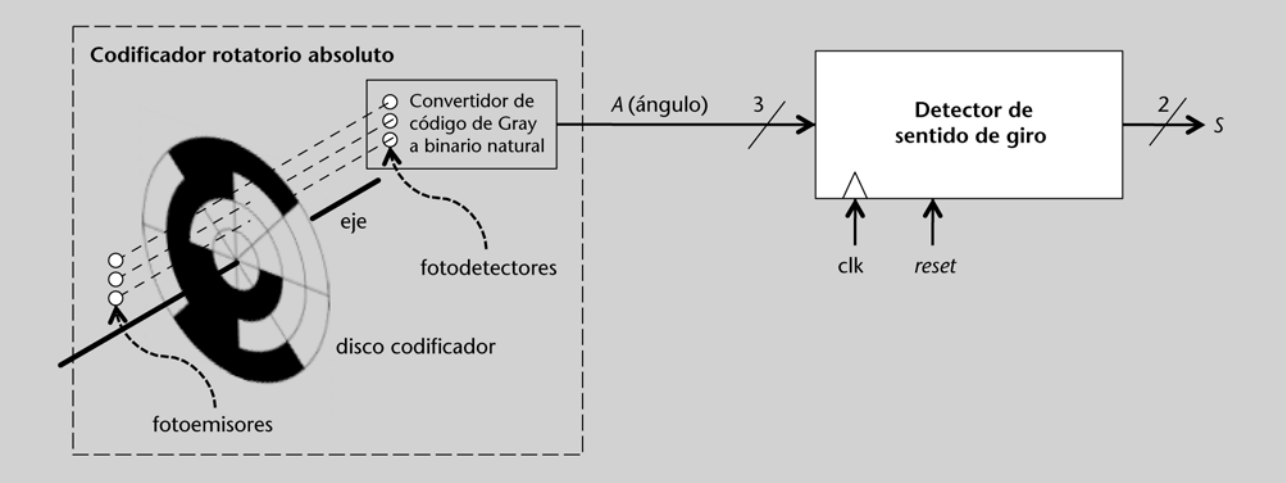
Códigos de Gray

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 [315º, 360º) se pasa directamente al sector [0º, 45º), 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



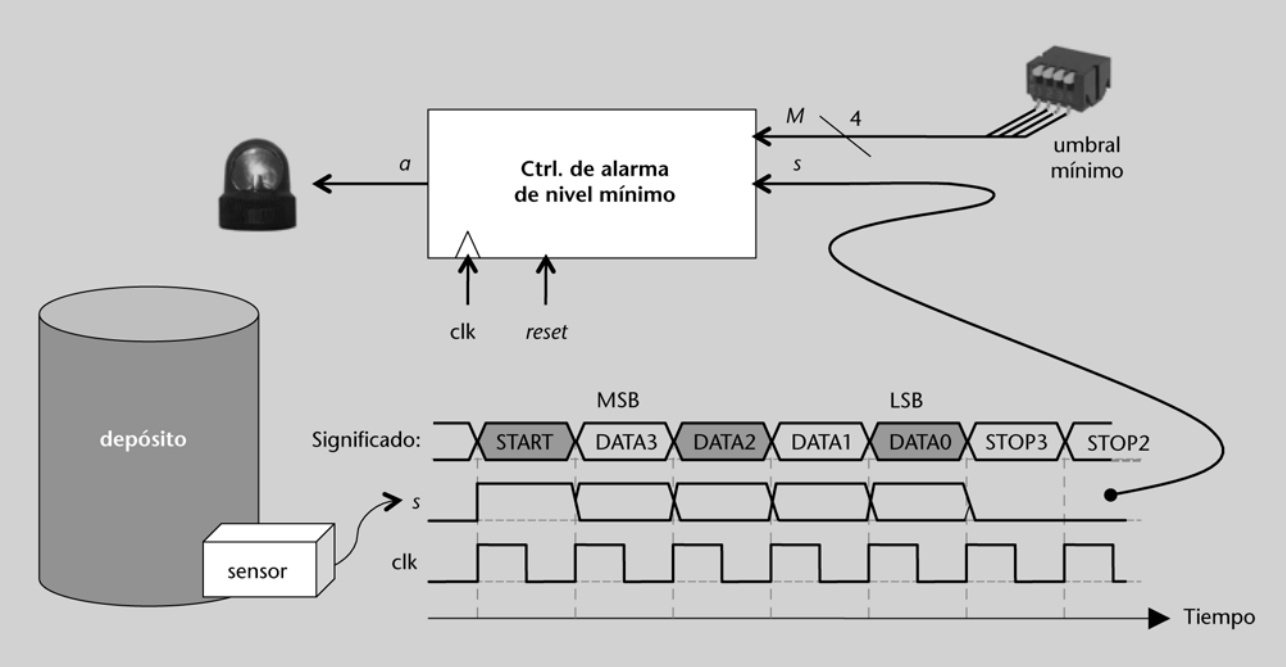
Para resolver este problema hay que tener presente que el detector de sentido de giro debe comparar el sector actual con el anterior. Se supone que la cadencia de las lecturas de los ángulos A es suficientemente elevada como para que no pueda haber saltos de dos o más sectores entre dos lecturas consecutivas.

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.

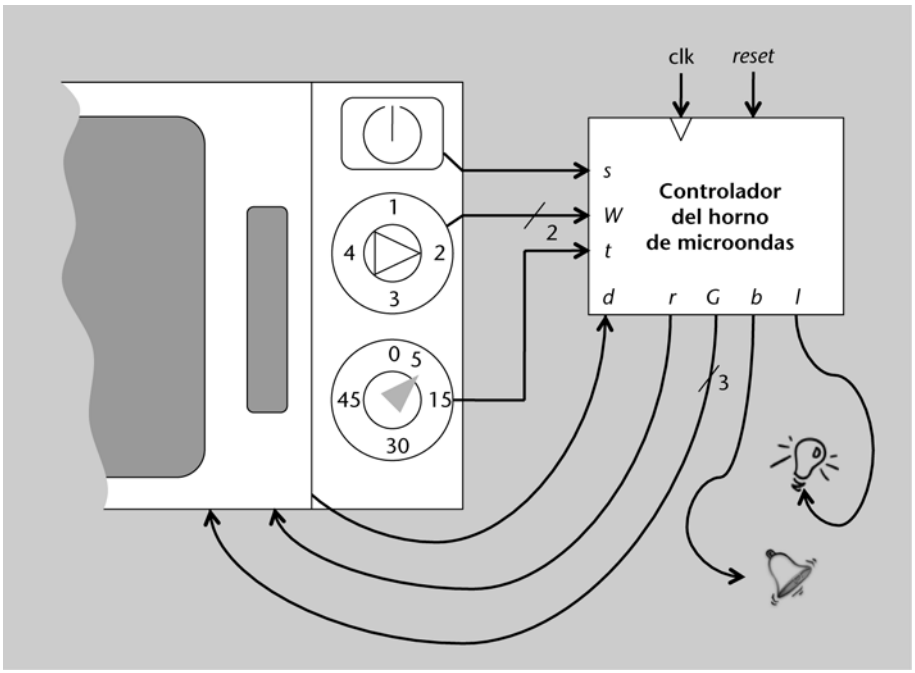
Hay que tener presente que el botón de inicio/parada se puede apretar en cualquier momento y que la rueda del temporizador solo puede volver a la posición 0 cuando haya pasado el tiempo correspondiente o si el usuario fuerza el retorno girándola hasta aquella posición. Si se aprieta para poner en marcha 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.

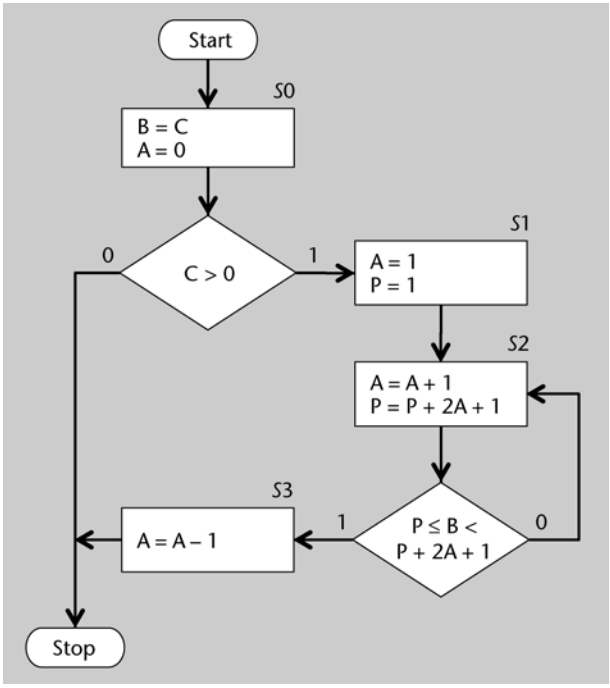
Para calcular a se puede seguir un procedimiento iterativo a partir de $a = 1$, incrementando gradualmente a hasta que cumpla la condición que se ha mencionado anteriormente.

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

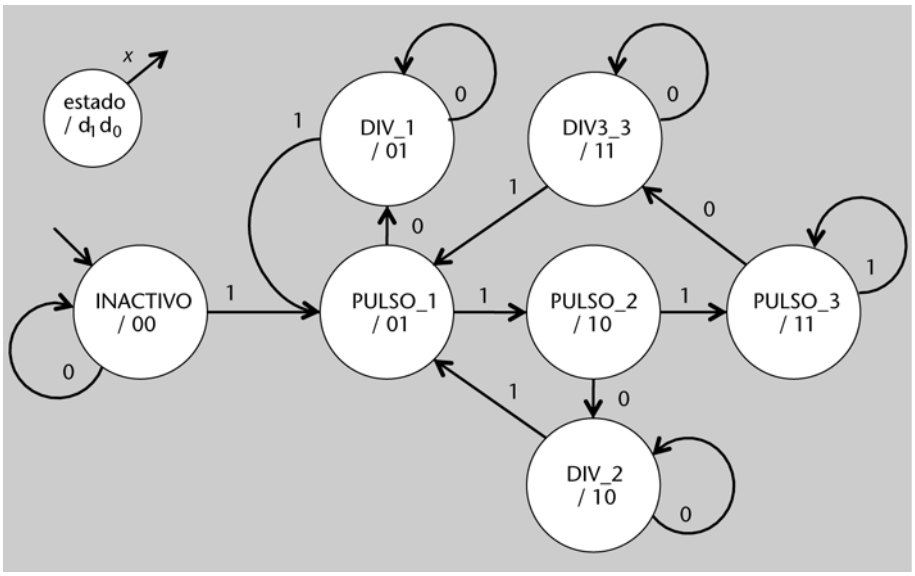
1. Para realizar el grafo de estados, hay que empezar por el estado inicial (INACTIVO) y decidir los estados siguientes, según todas las posibilidades de combinaciones de entradas. En este caso, solo pueden ser $x = 0$ o $x = 1$. Mientras no se detecte ningún cambio en la entrada ($x = 0$), la máquina 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

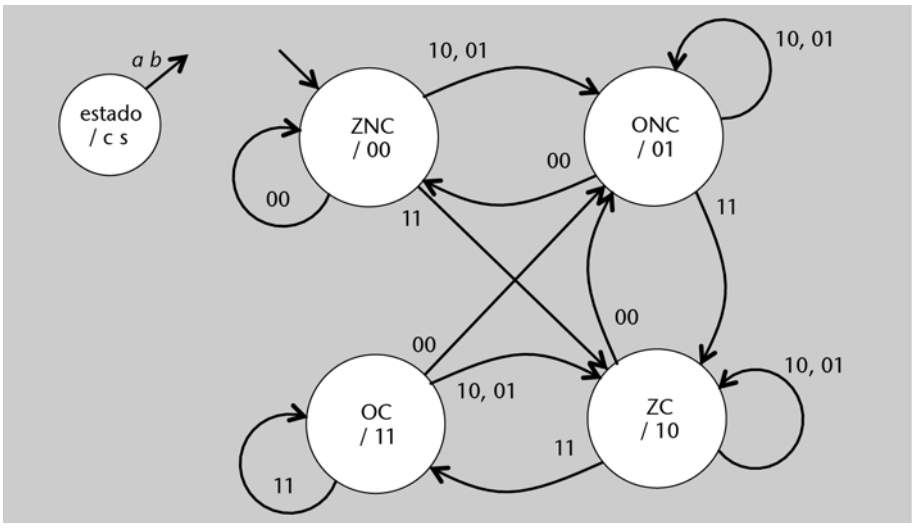
2. El estado de un sumador, en un momento determinado, lo fija tanto el resultado de la suma como del acarreo de salida del periodo de reloj anterior, que es el que hay que recordar. Por lo tanto, la máquina de estados correspondientes empieza en un estado en el que se supone que la suma previa ha sido (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

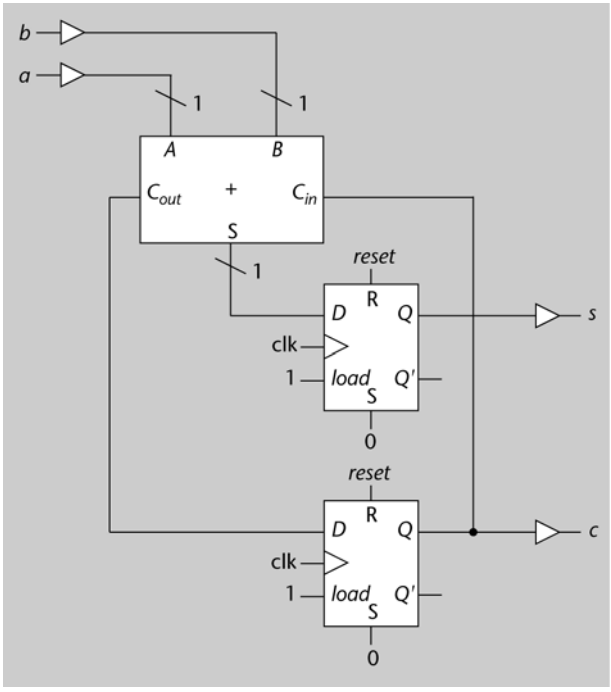
En este caso, la codificación de los estados se ha hecho respetando las salidas del sumador en serie. Por lo tanto, $c = q_1$ y $s = q_0$, como se puede comprobar en la tabla anterior.

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.

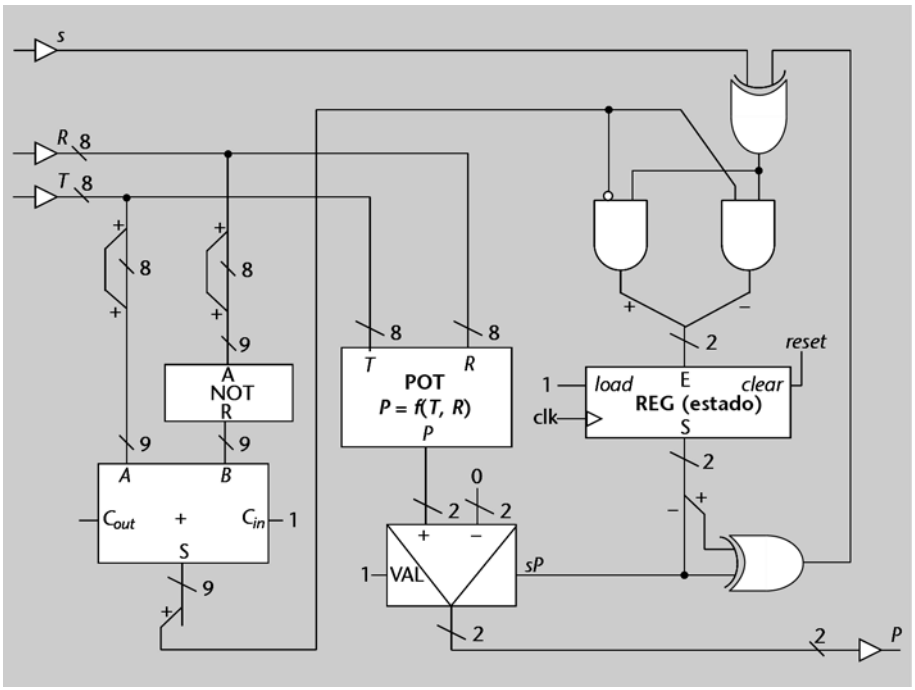
La codificación de los estados se realiza según el número binario correspondiente, teniendo en cuenta que el cero se reserva para el estado inicial (APAGADO). La codificación de la entrada de control será un bit para la señal de conmutación *s*. Y la condición ($T - R < 0$) se expresará con una señal 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.

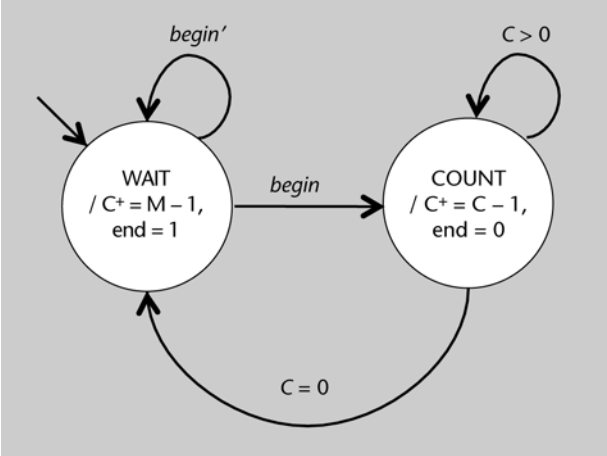
Figura 70. Controlador de un calefactor con termostato



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.

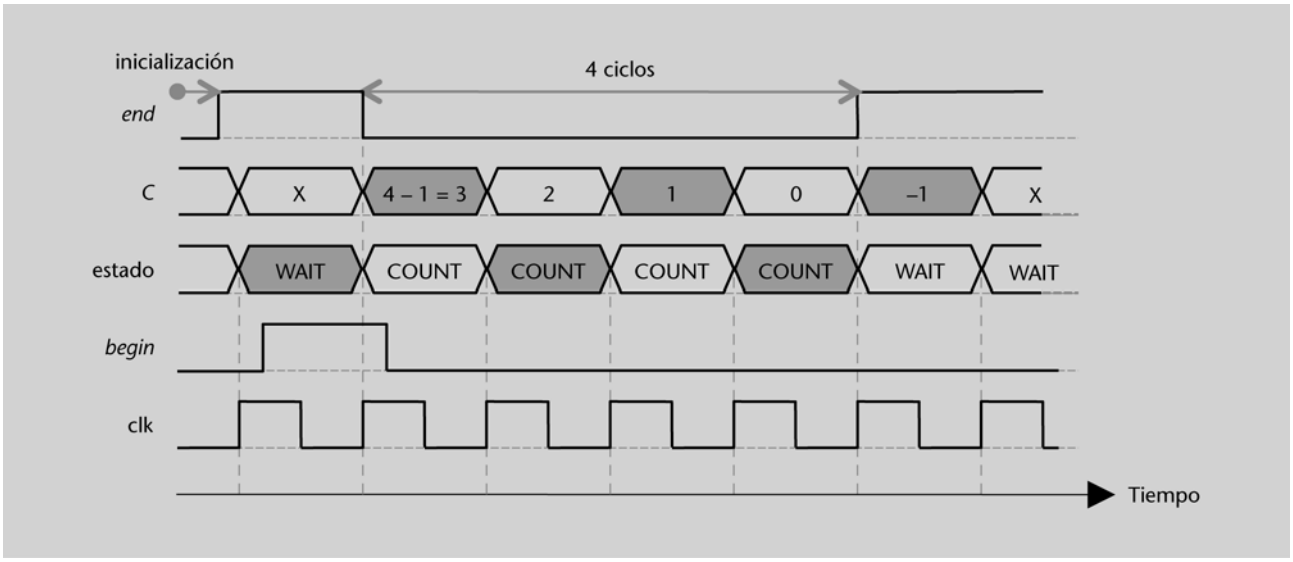
4. En este caso, en el estado de espera WAIT se deben ir efectuando cargas de valores decrementados ($M - 1$) en el registro de cuenta C hasta que *begin* = 1. Entonces se debe pasar al estado de ir contando, COUNT. En este estado, decrece C y, si el último valor de C ha sido cero, se acaba la cuenta. 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"



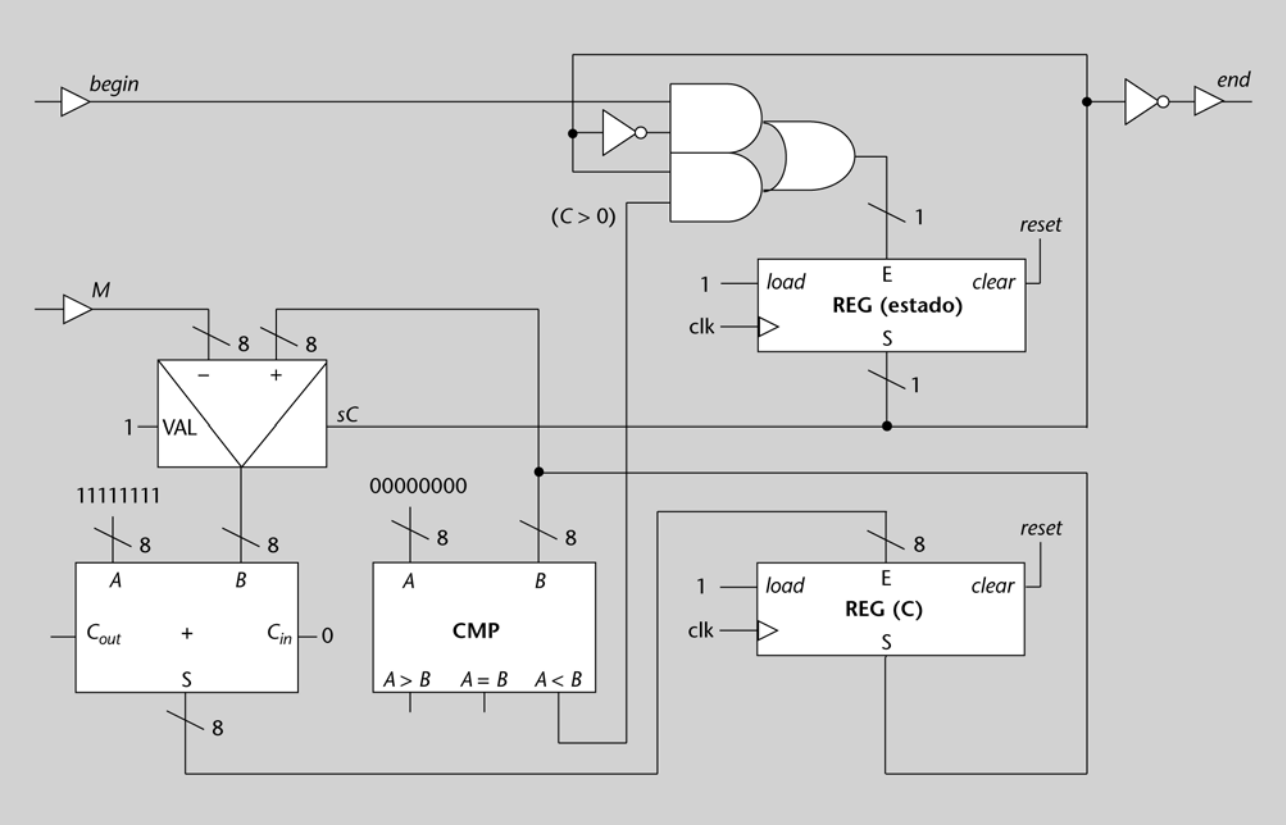
Hay que tener presente que, tal como sucede con el contador incremental, el registro C acaba cargando un valor de más que la cuenta que hace. En este caso, pasa a tener el valor –1 después de volver al estado WAIT. Este valor se pierde en el ciclo siguiente, con la carga de un posible nuevo valor M.

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.

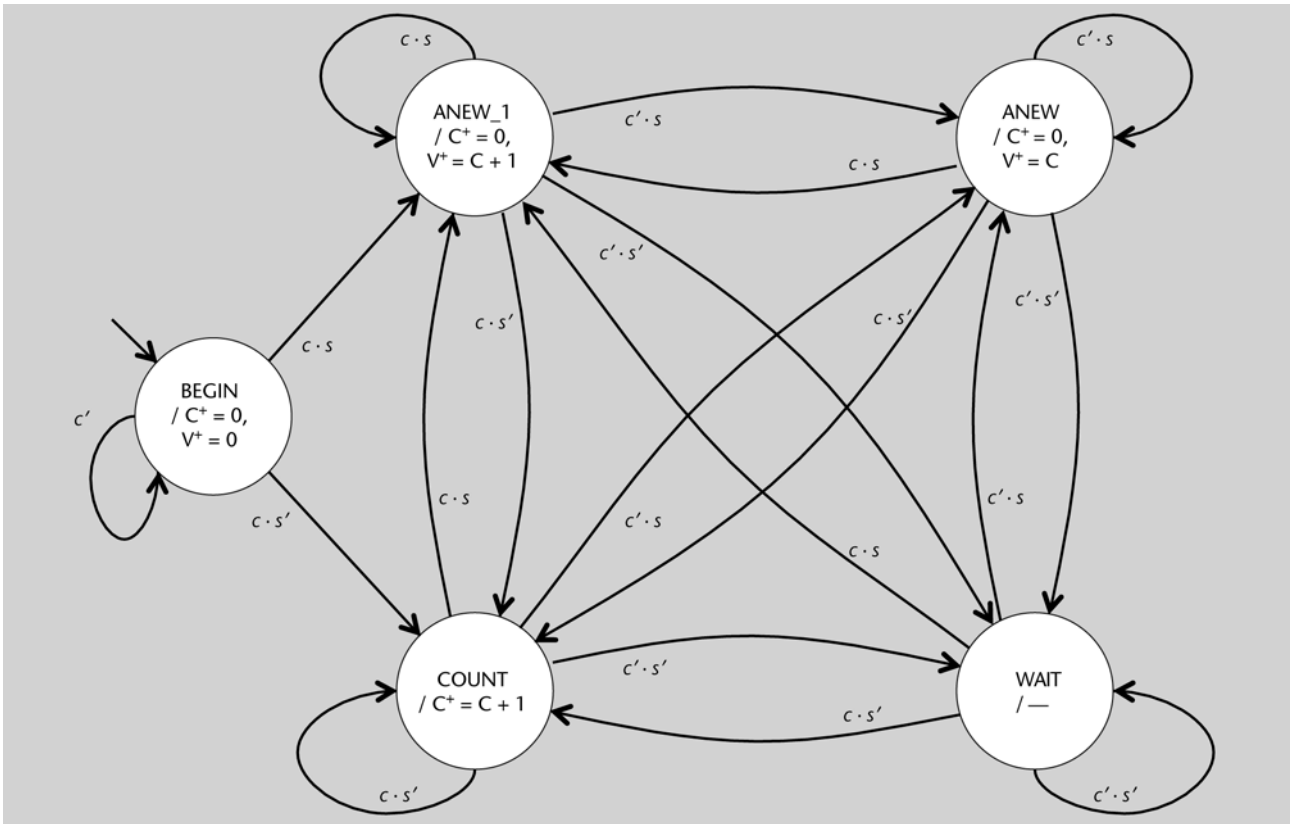
De hecho, cuando $s = 1$, habrá dos estados que reiniciarán la cuenta: ANEW, si no hay simultaneidad con c, y ANEW_1, si la hay. En el último caso se puede optar por inicializar el contador C a 1, que significa que el centímetro se contaría en el segundo posterior, o por acumularlo en la cuenta del segundo 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.

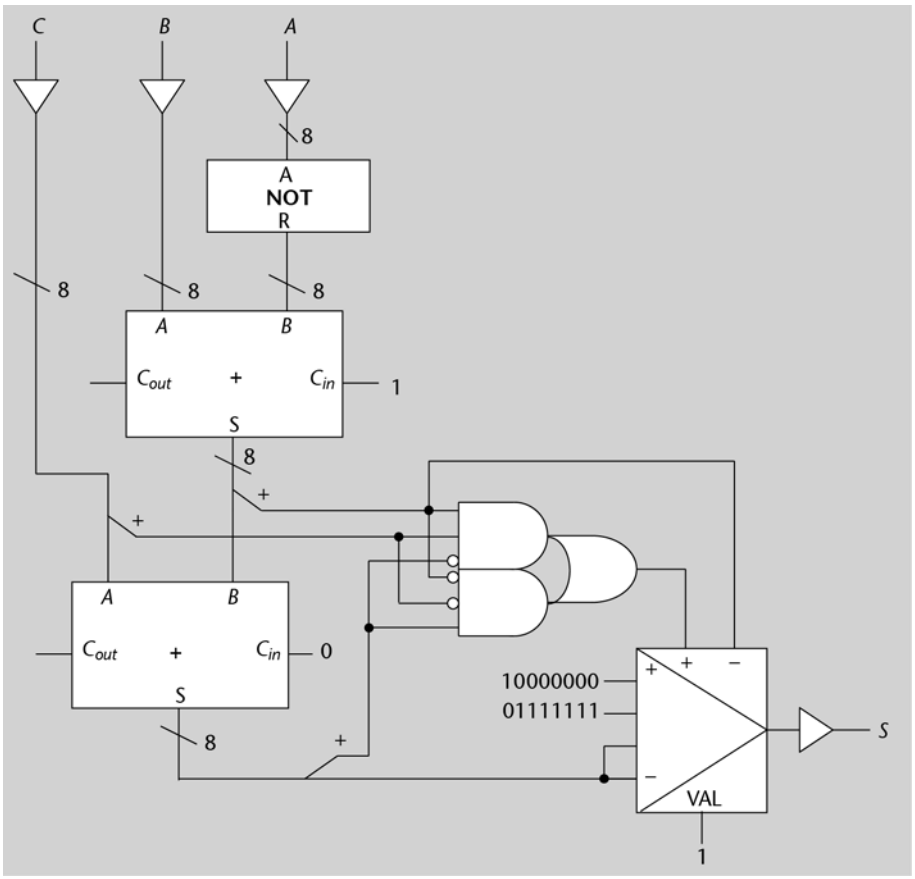
A modo de curiosidad, se podría utilizar el contador de la actividad anterior de manera que contara tantos ciclos de reloj como caben en un segundo. De este modo, el circuito conjunto solo necesitaría una entrada que le indicara cuándo se ha avanzado un centímetro, además de la de *reset* y de reloj, por supuesto.

Figura 74. Grafo de estados del velocímetro



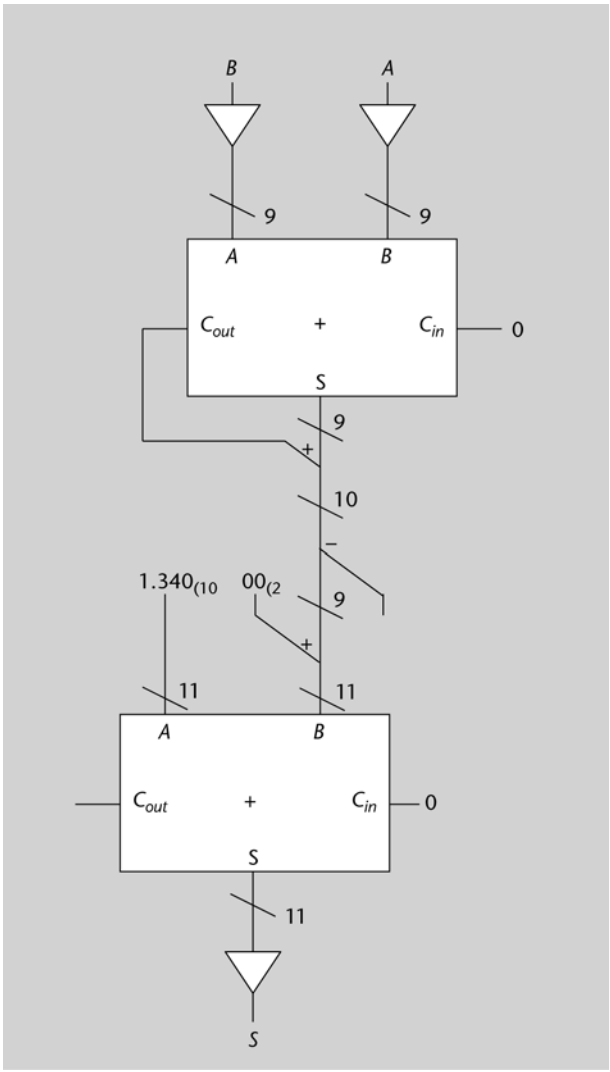
6. El módulo que calcula $C + B - A$ puede incurrir en desbordamiento. Para simplificar el diseño, primero se hace la resta, que nunca lo genera, y después la suma. En el caso de que se produzca desbordamiento, el resultado será el número mayor o el más pequeño posible, según sea la suma de números positivos 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.

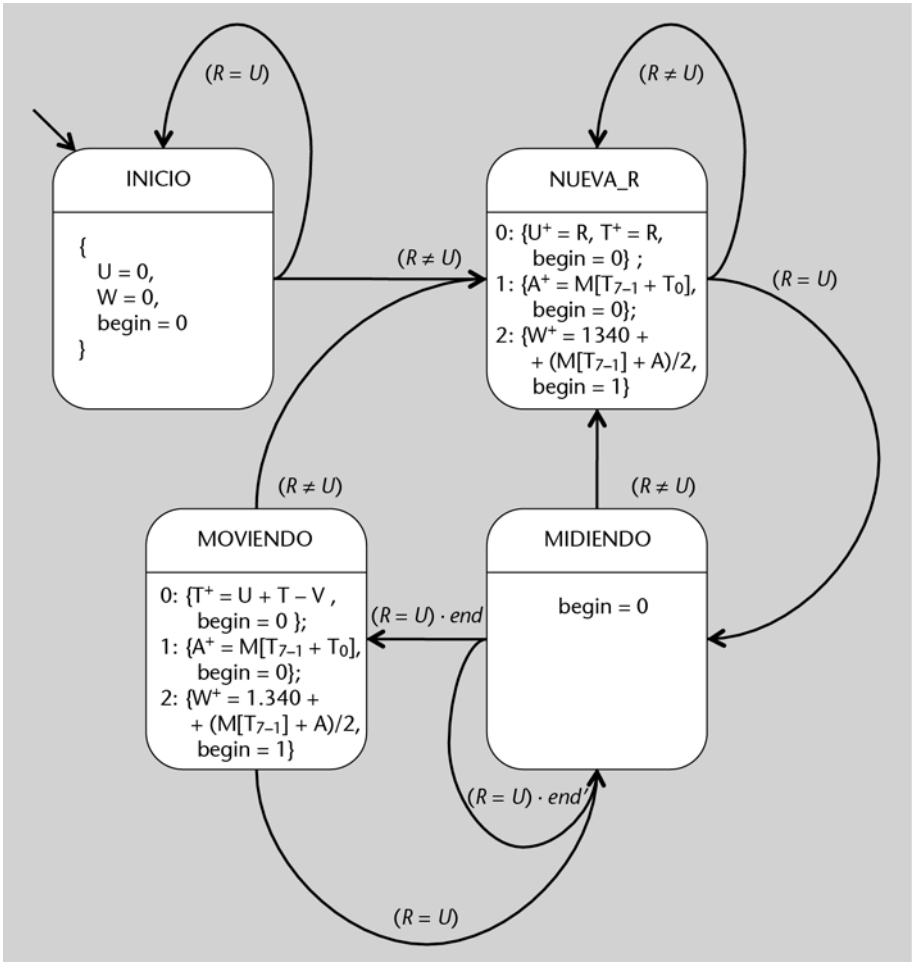
Figura 76. Módulo de cálculo de la amplitud de pulso del controlador de velocidad



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

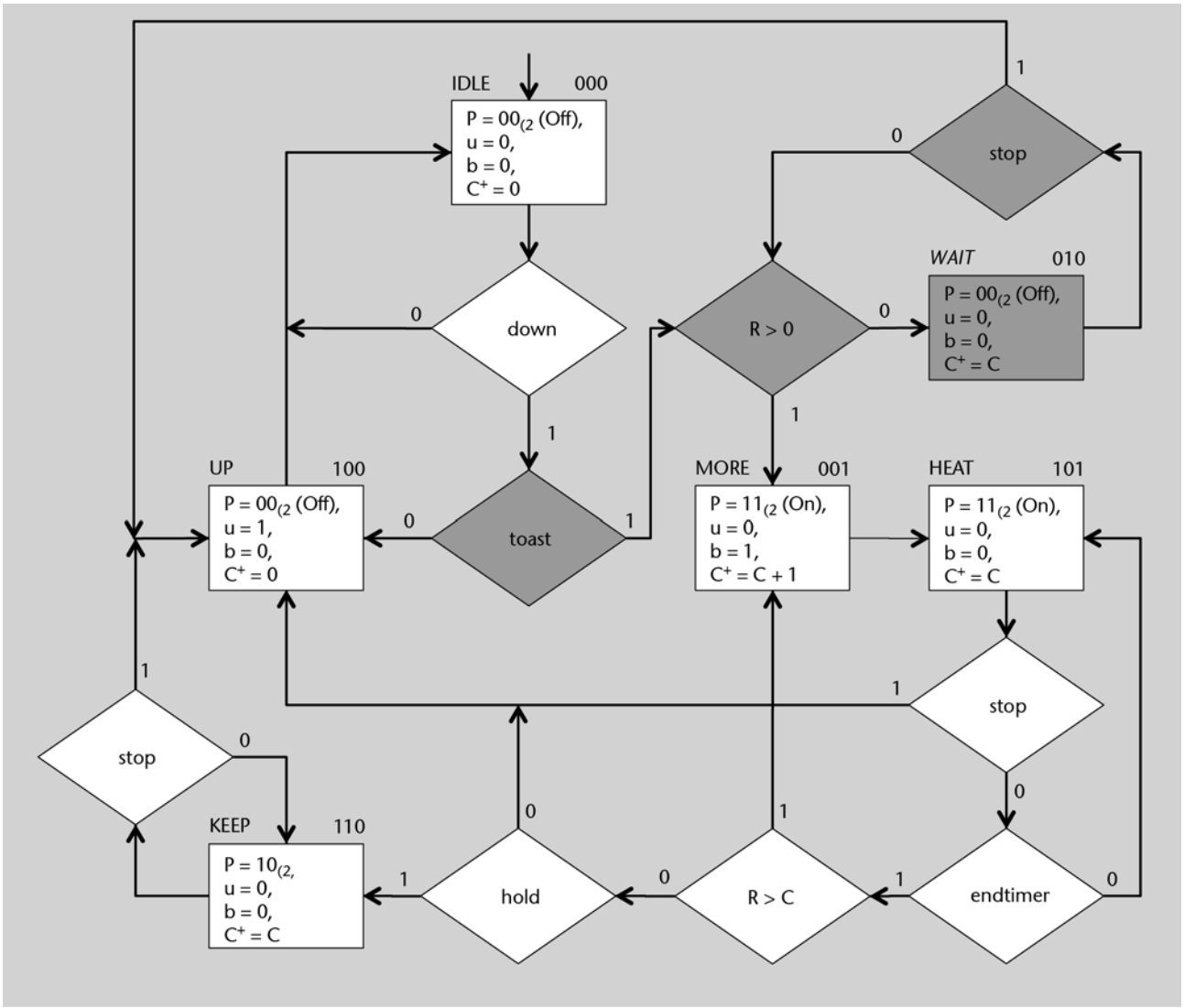
Se puede comprobar que existen muchos *don't-cares* que pueden ayudar a obtener expresiones mínimas para las funciones de cálculo del estado siguiente, pero también que son difíciles de manipular de manera manual.

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



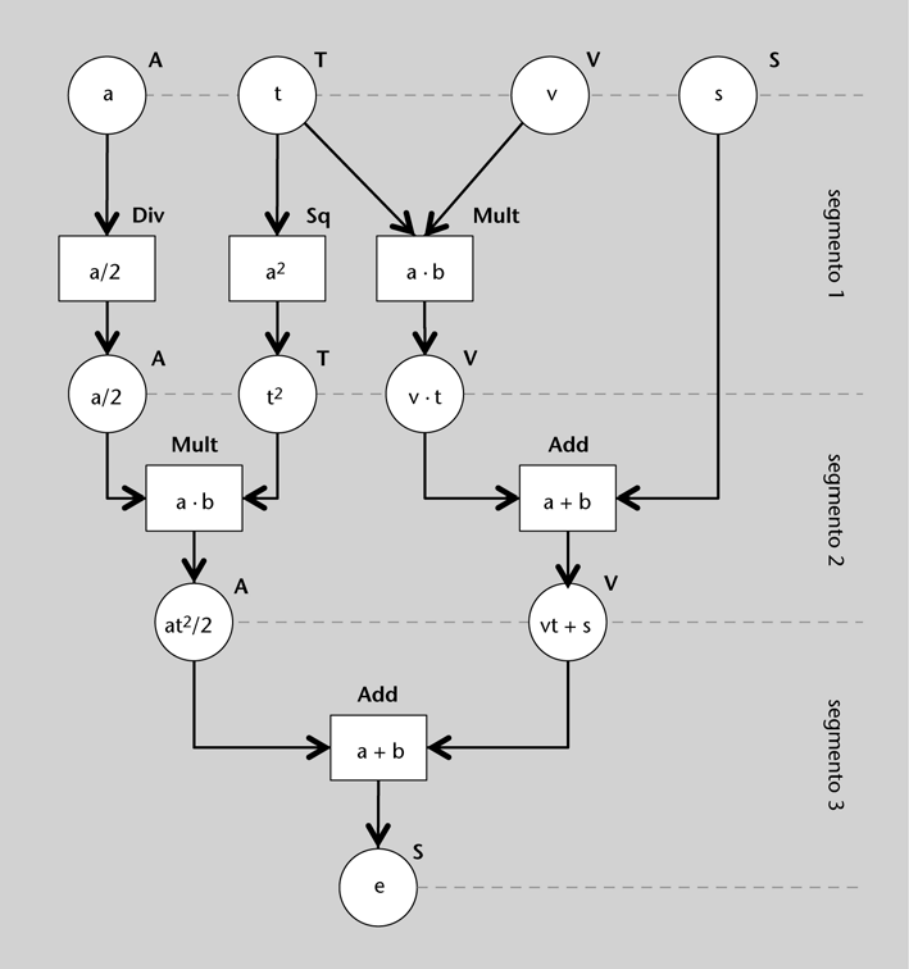
10. Antes de realizar el esquema de cálculo, es conveniente poner paréntesis a la expresión para agrupar las operaciones. Así, la expresión dada se puede transformar en:

$$(((a/2) \cdot (t^2)) + ((v \cdot t) + s))$$

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*.

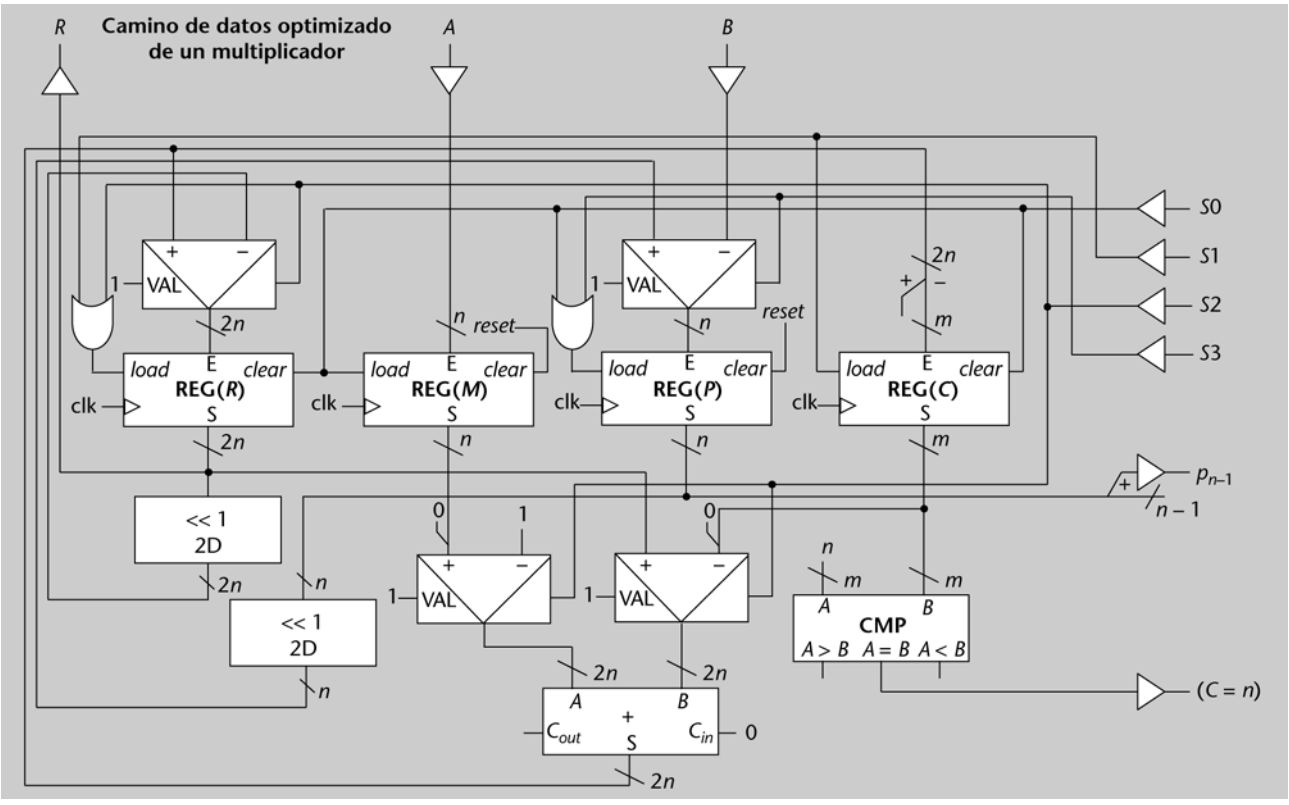
Finalmente, hay que tener en cuenta que las funciones lógicas que generan los valores de control de los multiplexores, de los *loads* y de los *reset* suelen ser más simples con la codificación *one-hot bit* que con la binaria. De hecho, en este caso no hay demasiada diferencia, pero utilizar la codificación *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. Pero, si se tienen en cuenta los módulos de memoria y de comunicaciones con el exterior, los registros MB y A tienen dos entradas cada uno y serían necesarias unas señales de un bit, *SelMB* y *SelA*, respectivamente, que servirían per escoger entre la entrada de los módulos externos y la del *busW*.

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.

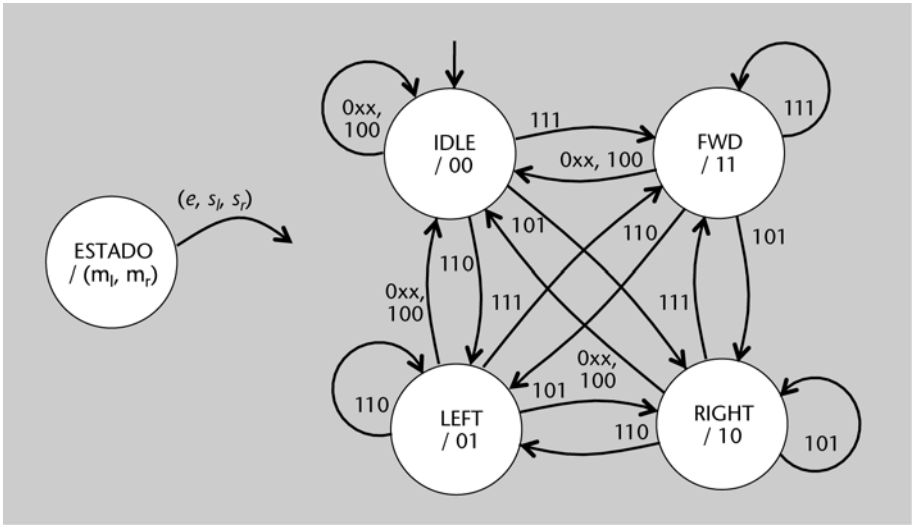
Ejercicios de autoevaluación

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.

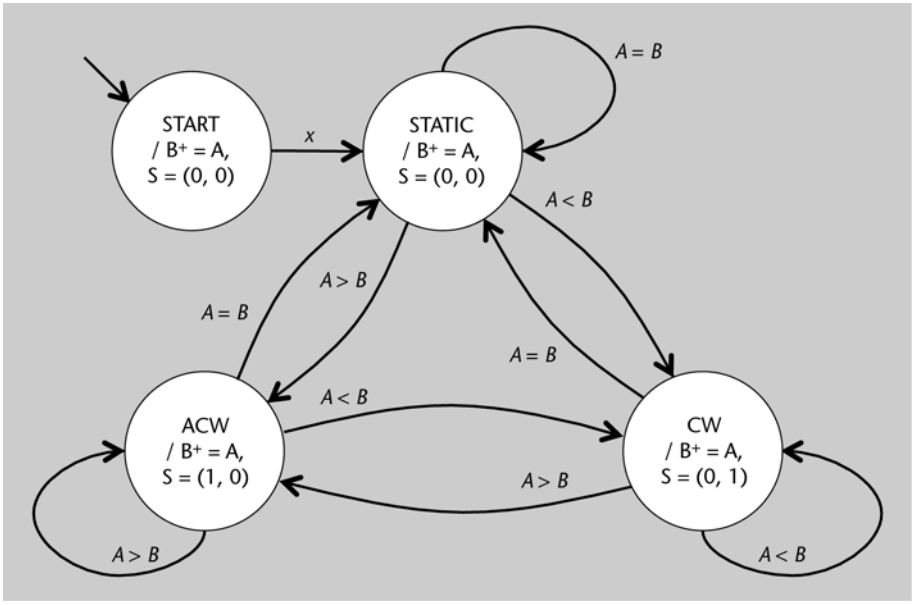
Figura 82. Grafo de estados del controlador de un seguidor de líneas



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.

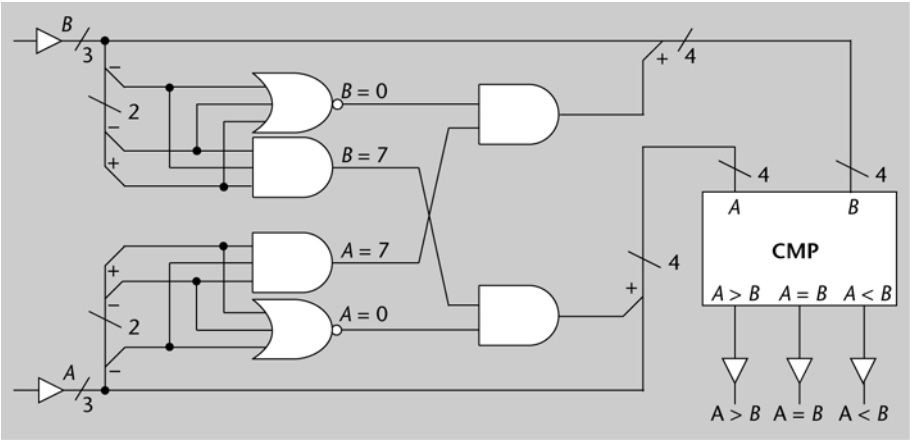
Estado actual			**Entradas**			**Estado siguiente**		
Identificación	q_1	q_0	$A > B$	$A = B$	$A < B$	**Identificación**	q_1^+	q_0^+
START	0	0	x	x	x	STATIC	1	1
CW	0	1	0	0	1	CW	0	1
CW	0	1	0	1	0	STATIC	1	1
CW	0	1	1	0	0	ACW	1	0
ACW	1	0	0	0	1	CW	0	1
ACW	1	0	0	1	0	STATIC	1	1
ACW	1	0	1	0	0	ACW	1	0
STATIC	1	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.

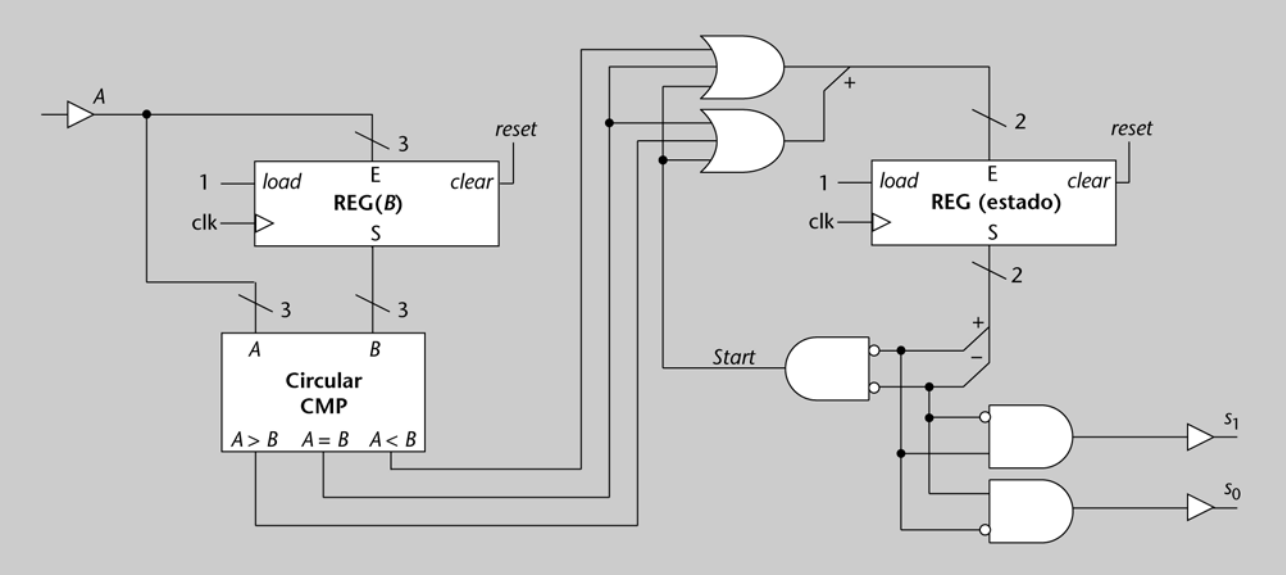
Las funciones de transición se obtienen de la tabla de verdad correspondiente. En este caso, basta con observar que el estado de destino depende exclusivamente de las entradas y que solo hay una entrada activa en cada momento, con la excepción de la transición de salida del estado START.

$$q_1^+ = q_1' \cdot q_0' + (A = B) + (A > B)$$
$$q_0^+ = q_1' \cdot q_0' + (A = B) + (A < B)$$

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:

$$L^+ = (L << 1) + s$$

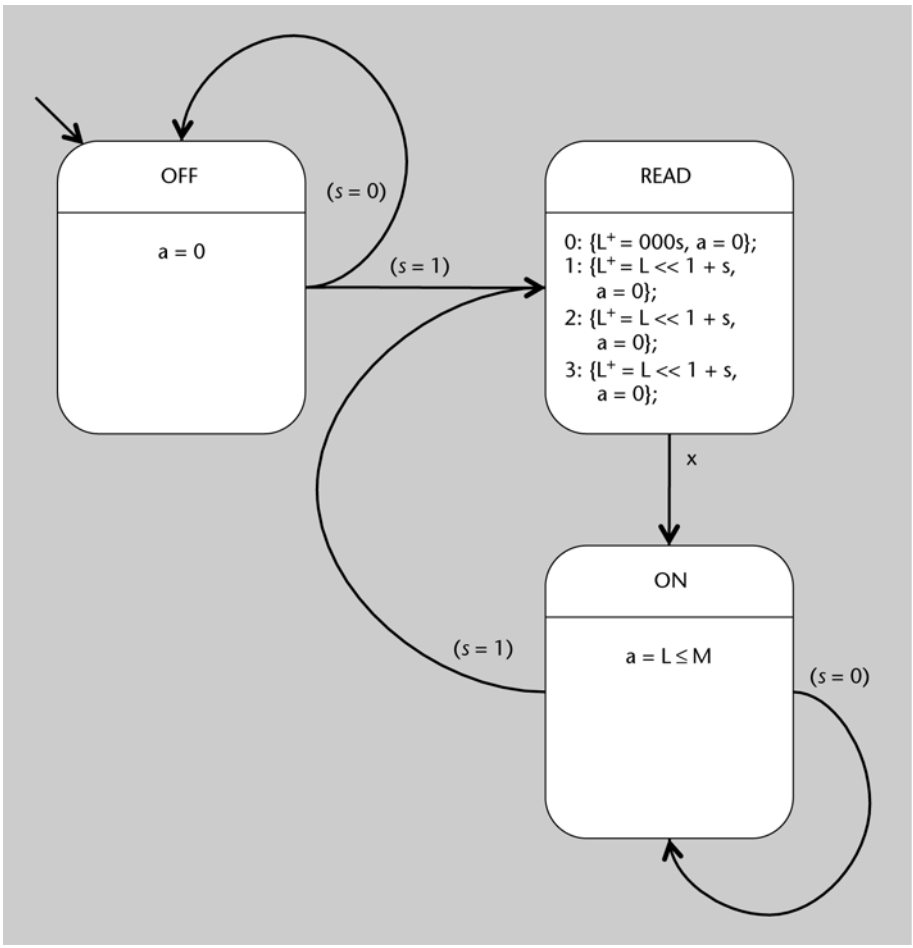
De esta manera, después de 3 pasos, el bit más significativo ya está en la posición más a la izquierda, y el menos significativo, en la posición de las unidades. Por simplicidad, se ha optado por que, durante la ejecución del programa de lectura, la alarma se detenga. En la práctica, esto no debería 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:

$$a = \text{ON} \cdot (L \leq M)$$

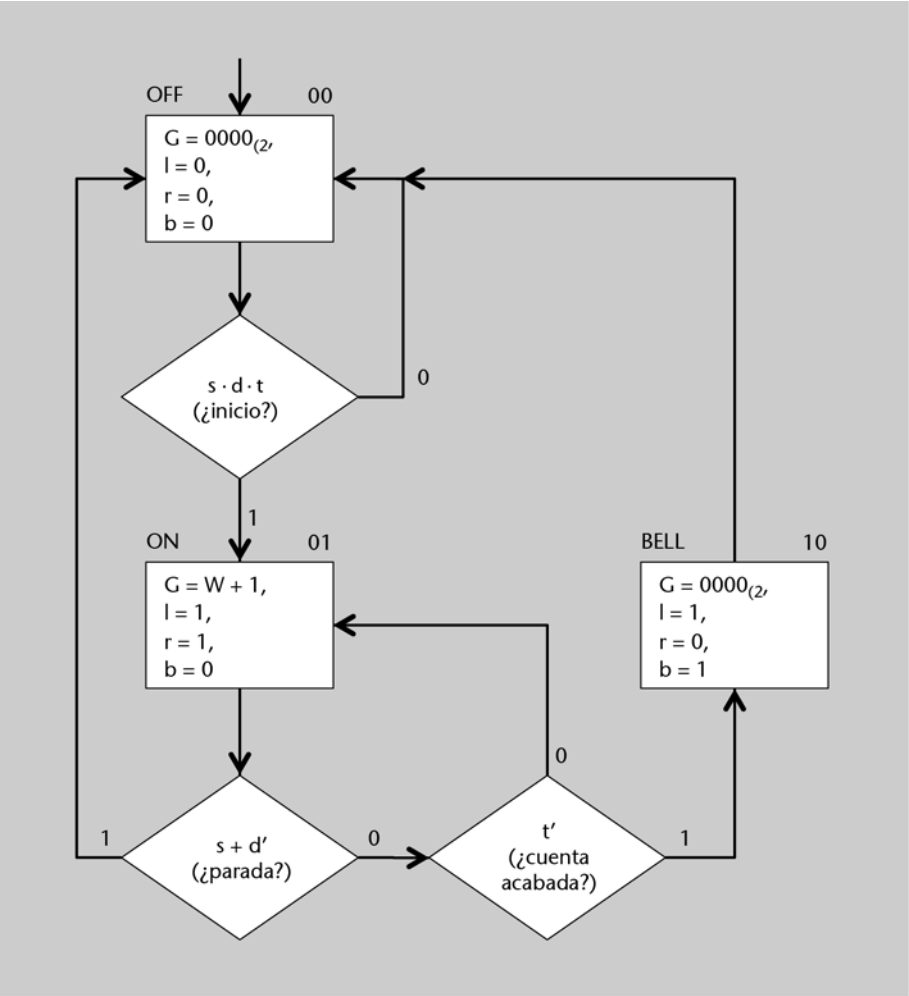
En la unidad de procesamiento debería estar el registro *L* y los módulos combinacionales necesarios para realizar las operaciones de decalado ($L << 1$), suma y comparación. Hay que tener presente que se debería seleccionar qué valor se asigna a *L* con un multiplexor y que, por lo tanto, es necesaria 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.

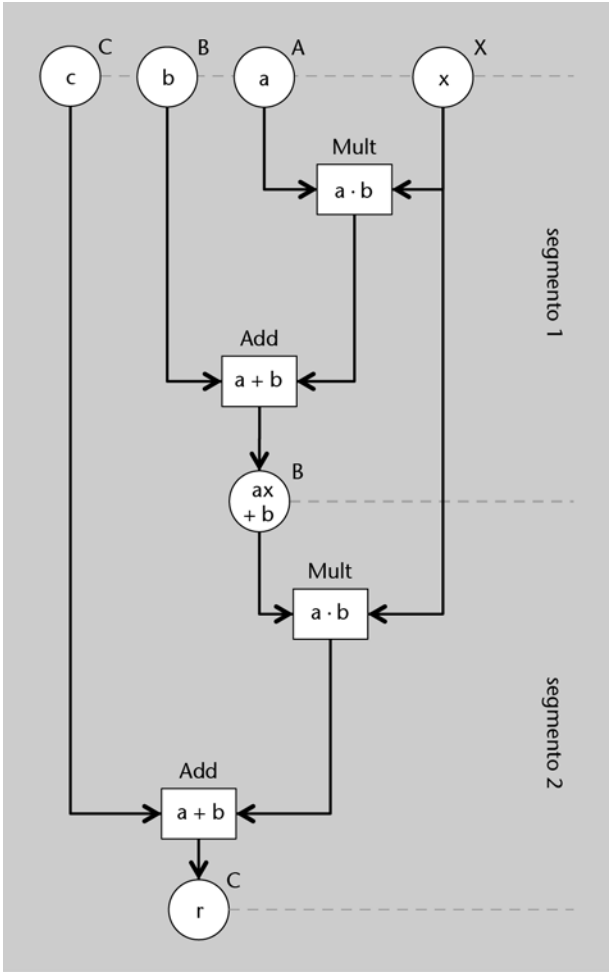
5. Como se trata de realizar el cálculo con el mínimo número de recursos, es necesario minimizar el número de operaciones y aprovechar los registros que almacenan las variables de entrada para resultados intermedios y para el final. En cuanto al primer punto, se puede reescribir la expresión en la forma:

$$(ax + b)x + c$$

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 $S3$ se reduce A en una unidad porque la actualización de los registros A y P que se hace en el estado $S2$ 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 $S2$. Por lo tanto, al llegar a $S3$, 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:

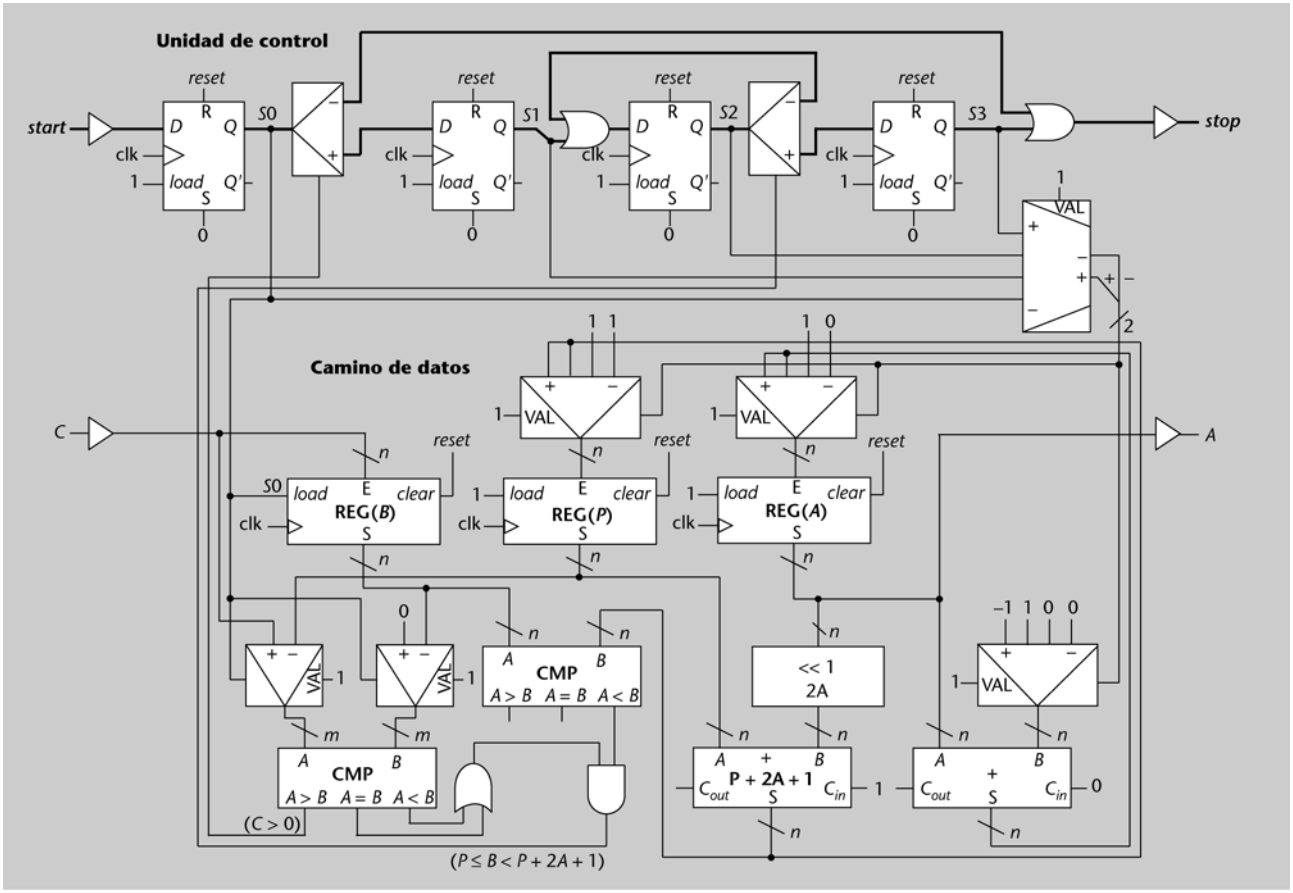
- Se ha eliminado el multiplexor en la entrada del registro B porque este registro solo carga un valor $S0$ (el contenido de la señal de entrada C).

- 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 $S0$.

- 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 $S2$.

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

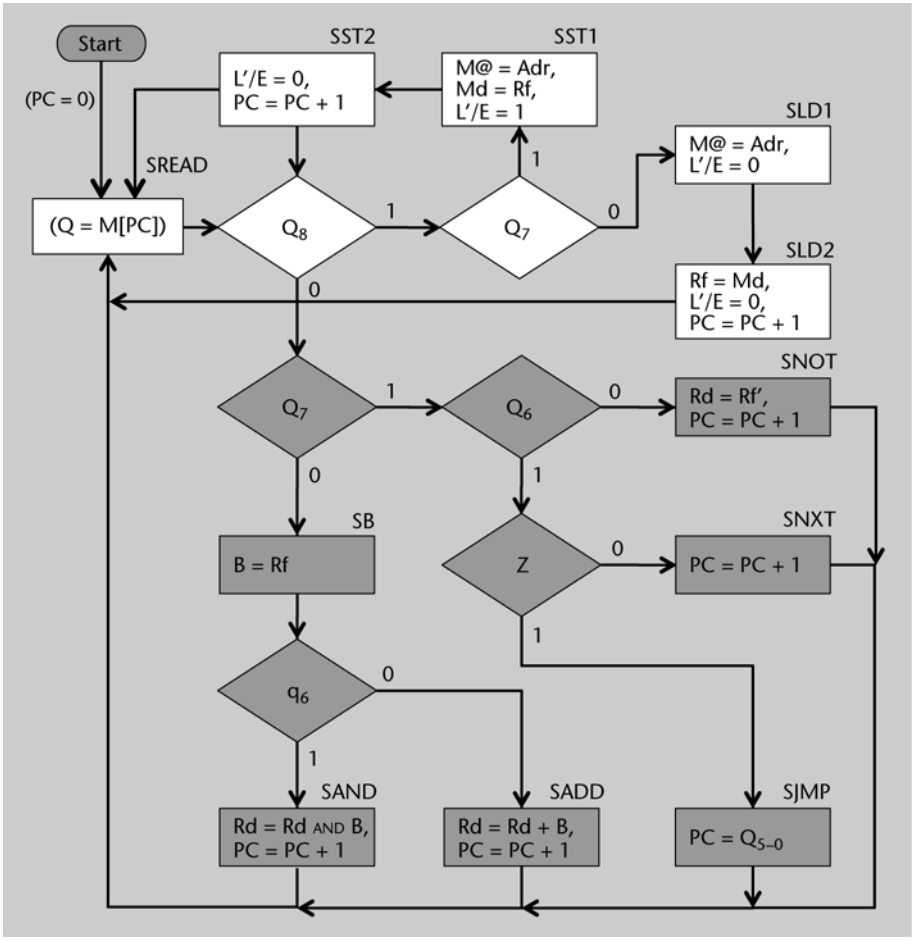


7. El algoritmo de interpretación no cambia para las instrucciones ya existentes en el repertorio del Femtoproc, solo hay que añadir la parte correspondiente a la interpretación de las nuevas instrucciones. Para hacerlo, en el diagrama de flujo correspondiente se debe poner un nodo de decisión justo 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.