Estructura básica de un computador

El procesador como generalización de las máquinas de estados algorítmicos

PID_00279131

Lluís Ribas i Xirgo

Tiempo mínimo de dedicación recomendado: 9 horas

Universitat Oberta de Catalunya

Lluís Ribas i Xirgo

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Javier Panadero Martínez

Primera edición: febrero 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Lluís Ribas i Xirgo
Producción: FUOC

Índice

Introducción

En el módulo «Circuitos lógicos secuenciales» se ha visto que los circuitos secuenciales sirven para detectar el orden temporal de una serie de hechos (es decir, secuencias de bits) y que también, a partir de las 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 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 respectivamente las máquinas de estados y los bloques de procesamiento de datos.

En este módulo se aprenderá a abordar 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 hacer el diseño de las unidades de procesamiento y de control por separado. Lo mismo pasa en casos algo más complejos en que la frase anterior empieza con «si se cumple» o «mientras se cumpla», o si los cálculos implican bastantes pasos.

Finalmente, esto ayudará a comprender cómo se construyen y cómo trabajan los procesadores, que son el componente principal de cualquier computador.

El módulo termina mostrando las arquitecturas generales de los computadores de forma que sirva de base para comprender el funcionamiento de estas máquinas.

Objetivos

La finalidad de este módulo es que se aprendan los conceptos fundamentales para el análisis y la síntesis de los diversos componentes de un computador visto como un sistema digital secuencial complejo. A continuación, se enumeran los objetivos particulares que se deben alcanzar una vez acabado el estudio de este módulo para cumplir con la finalidad que se ha enunciado:

1. Conocer los diversos modelos de las 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 de estados algorítmicas y entender los criterios que lo afectan.

6. Ser capaz de diseñar máquinas de estados algorítmicas como modelos de programas sencillos.

7. Haber aprendido el funcionamiento de los procesadores como máquinas de estados 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.

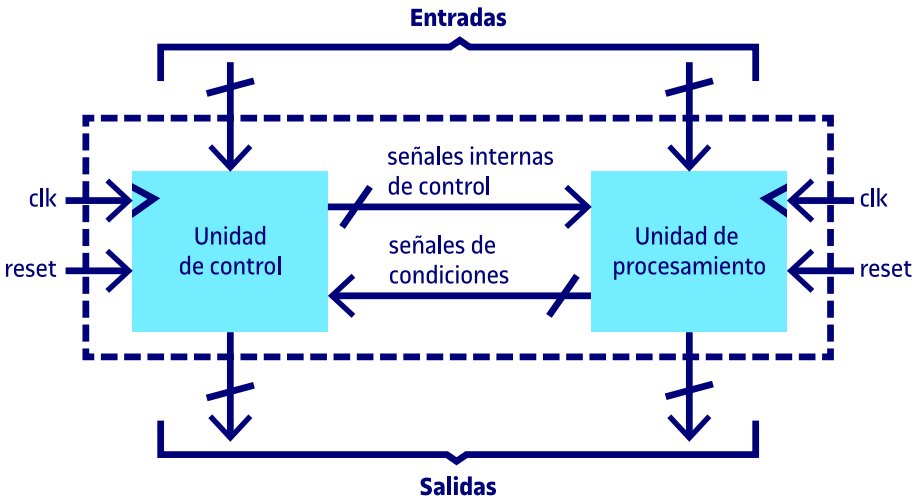
1. Máquinas de estados

Los circuitos secuenciales son máquinas de estados que tienen un comportamiento que 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 **unidad de control**, que se ocupa de las transiciones de estados y que implementan las funciones de excitación de la máquina de estados correspondiente, y la **unidad de procesamiento**, que se ocupa de hacer las operaciones con los datos tanto para determinar condiciones de transición como para calcular resultados de salida.

La figura 1 ilustra esta organización. La unidad de control es una máquina de estados que recibe información del exterior a través 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 haga unos cálculos determinados. Estos cálculos se hacen con datos del exterior y también internos, puesto que la unidad de procesamiento también dispone de elementos de memoria. Los resultados de estos cálculos pueden ser observables desde el exterior y, por eso, esta unidad también tiene señales de salida.

Figura 1. Organización de un circuito secuencial



En el módulo «Circuitos lógicos secuenciales» se ha explicado cómo funcionan los circuitos secuenciales que materializan una máquina de estados con poco o ningún procesamiento. En este capítulo se verá cómo relacionar los estados con las operaciones, de forma que la representación de la máquina de estados también incluya 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 un elemento con capacidad de actuar sobre otro para llevarlo a un estado determinado.

El regulador de potencia de un calefactor es un controlador de la intensidad del radiador. Cada posición del regulador (por ejemplo: 0 para apagado, 1 para media potencia y 2 para potencia máxima) es un estado del controlador y también un objetivo para la parte que se controla. Así, se puede decir que la posición 1 del regulador quiere decir que hay que llevar el radiador al estado de consumo de una intensidad mediana de corriente. El mismo ejemplo ilustra la diferencia con una máquina de estados no finitos: si el regulador consistiera en una rueda que se pudiera girar hasta situarse en cualquier posición entre 0 y 2, entonces haría falta un número (teóricamente) infinito de estados.

Otro ejemplo de controlador es el del mecanismo de llenado de agua del depósito de una taza de wáter. En este caso, el sistema que hay que controlar tiene dos estados: el depósito puede estar lleno (LLENO) o puede no estarlo (NO_LLENO).

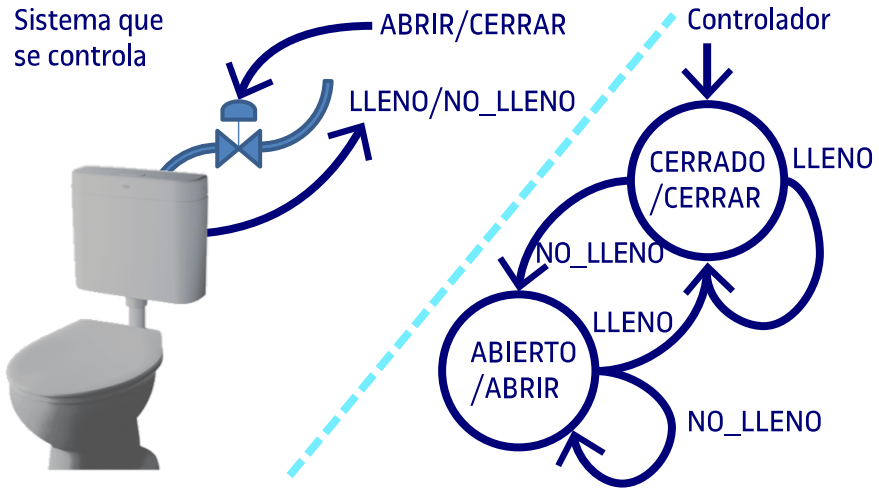
El mecanismo de control de llenado de agua se ocupa de llevar el sistema al estado de LLENO de manera autónoma. En otras palabras, el controlador tiene una referencia interna del estado al que tiene que llevar el sistema que controla. Para hacerlo, tiene que detectar el estado y, con este dato de entrada, determinar qué acción debe hacer: abrir el grifo de llenado (ABRIR) o cerrarlo (CERRAR). Además, hay que tener presente que el controlador tendrá que mantener el grifo abierto mientras el depósito no esté lleno, y cerrado siempre que esté lleno. Esto hace que tenga que «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 (figura 2).

Siguiendo con el ejemplo, los estados captados son LLENO y NO_LLENO, las acciones son CERRAR y ABRIR, y el objetivo para el sistema que se controla, que esté LLENO. El estado inicial del controlador de llenado de la cisterna tiene que ser CERRADO, que está asociado a la acción de CERRAR. De este modo, al empezar el funcionamiento del controlador queda garantizado que no habrá desbordamiento del depósito. Si ya está LLENO, se tiene que quedar en el mismo estado. Pero, si se detecta que está NO_LLENO, se tiene que llenar. Para lo cual tiene que pasar al estado de ABIERTO, en que se abre el grifo. El grifo se mantiene abierto hasta que el depósito esté LLENO. En este momento, se tiene que CERRAR y se pasa 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

Estado del controlador	Acción
Estado actual	Salida
CERRADO	CERRAR

La **materialización** del controlador de llenado del depósito de agua de una taza de wáter se puede resolver muy eficazmente sin ningún circuito secuencial: la mayoría de cisternas llevan una válvula de admisión de agua controlada por un mecanismo con una boya que flota. Con todo, aquí se hará con un circuito digital que implemente la máquina de estados correspondiente, un sensor de nivel y una electroválvula, para lo cual, toda la información se codifica en binario.

En este caso, la observación del estado del sistema (la cisterna del wáter) se hace cogiendo los datos del sensor, que son binarios: LLENO (1) o NO_LLENO (0). Las actuaciones del controlador se hacen a partir 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 hacer el circuito secuencial del controlador del depósito, hay que hacer 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

Tal 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 hará referencia como **entradas**.

1.2. Materialización de controladores con circuitos secuenciales

Para diseñar un circuito secuencial que implemente una máquina de estados correspondiente a un controlador hay que seguir los pasos siguientes:

- Definición de las salidas
- Definición de las entradas
- Diseño de la máquina de estados
- Codificación en binario de las señales y de los estados
- Diseño de los circuitos

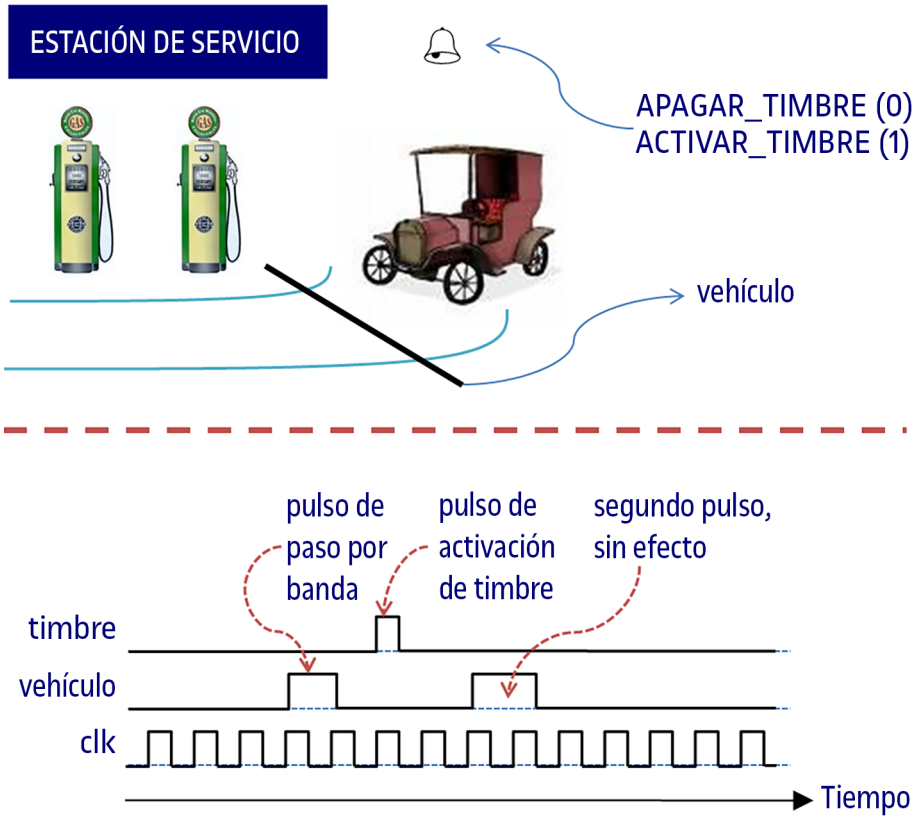
Para ilustrarlos, se ejemplificarán con el caso de un controlador que hace sonar un timbre de aviso de entrada de vehículos en una estación de servicio (figura 3).

1.2.1. Definición de las salidas del controlador

De acuerdo con la funcionalidad que se debe implementar, hay que decidir qué acciones se tienen que hacer 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, basta con enviar un pulso a 1 al aparato correspondiente. Por simplicidad, se supone que solo tiene que durar un ciclo de reloj.

Figura 3. Sistema de aviso de entrada de vehículos a una estación de servicio



1.2.2. Definición de las entradas del controlador

Se trata de determinar qué información hay que obtener del sistema que se controla.

En cuanto al ejemplo, el controlador necesita saber si algún vehículo pasa por la vía de entrada, para lo cual se puede tener un cable sensible a la presión fijado al suelo. Si se activa, significa que un vehículo lo está pisando y que hay que 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») diferentes, 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 por ejemplo un camión, el timbre sonaría más de una vez. Con todo, estos casos no se tienen en cuenta para el ejemplo.

1.2.3. Diseño de 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 una estación de servicio.

Para ello, se empieza con un estado inicial de reposo en que se decide hacia qué estados tiene que pasar según todas las posibles combinaciones de entrada, y para cada uno de dichos 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 tiene que permanecer 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 hay que 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 más grande que la velocidad de paso de cualquier vehículo. Por lo tanto, la detección de un pulso se hace al completar una secuencia del tipo 01...10, con un número de bits indefinido entremedio.

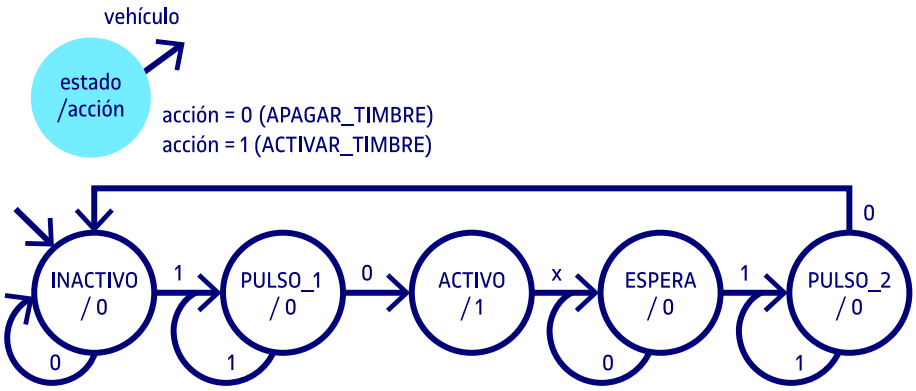
Así pues, la máquina pasa al estado PULSO_1 después de haber detectado un flanco de subida (paso de 0 a 1) en *vehículo*, y no tiene que salir hasta que haya un flanco de bajada que indique la finalización del primer pulso. En este momento, tiene que pasar a un estado en que se active el timbre, ACTIVO.

El estado ACTIVO es el único asociado a la acción de ACTIVAR_TIMBRE. Dado que 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. Ese nuevo estado esperará el pulso correspondiente al paso del segundo eje del vehículo. De hecho, harán falta 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 fin del pulso.

Habiendo acabado del segundo pulso, hay que pasar de nuevo al estado inicial, INACTIVO. De este modo, se queda en espera de que venga otro vehículo.

Con esto ya se puede construir el diagrama de estados correspondiente (figura 4). 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» posibles del sistema que se controla) en cada estado.

Figura 4. Grafo de estados del sistema de aviso de entrada de vehículos



Con vistas 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 hace sin importar el valor de *vehículo*, lo 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. Sin embargo, es difícil que se dé este caso cuando la velocidad de muestreo es mucho más elevada que la de cambio de valores en *vehículo*. Con todo, también se puede alterar el diagrama de estados para que no sea posible (se puede hacer como ejercicio voluntario).

1.2.4. Codificación en binario de las señales de entrada y salida, y del estado del controlador

A pesar de que, 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 codificar los estados del controlador. Generalmente, se opta por dos políticas: tantos bits como estados diferentes y solo uno de activo para cada estado (en inglés: *one-hot bit*), o numerarlos con números binarios naturales, del 0 al número de estados que haya menos uno, tal como se puede ver en el ejemplo siguiente. En este último caso, habitualmente, el estado codificado con 0 es el estado inicial, puesto que facilita la implementación del *reset* del circuito correspondiente.

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 que solo el bit asociado es 1, es decir, el símbolo número i se asocia al código con el bit en posición i a 1. Por ejemplo, si hay dos símbolos, s_0 y s_1, la codificación de bit único sería 01 y 10, respectivamente.

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

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. Esto es un ejemplo de que la codificación binaria de los estados tiene un impacto directo en la materialización de los circuitos y, de hecho, es un problema en sí mismo, que queda fuera del ámbito de estudio de esta asignatura.

1.2.5. Diseño de los circuitos de la máquina de estados

El modelo de construcción de una FSM como controlador que toma señales binarias de entrada y 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.

A pesar de que la parte combinacional se separa en dos partes —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 a la estación de servicio, el circuito del controlador binario es el que se muestra en el esquema de la figura 6. En el circuito, la señal de entrada *v* es la que indica la detección de paso de eje de vehículos (*vehículo*) y *t* (*timbre*), la de salida que permite poner en marcha el timbre. Las expresiones de las funciones lógicas de cálculo del estado siguiente comparten un término (q_1q_0') que, además, se puede aprovechar como función de salida (*t*). Las entradas de los sensores están en el lado izquierdo, y las salidas hacia el actuador (el que hace la acción, que es el timbre), en el derecho.

Buffers

En los circuitos, a menudo se usan 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 usan 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

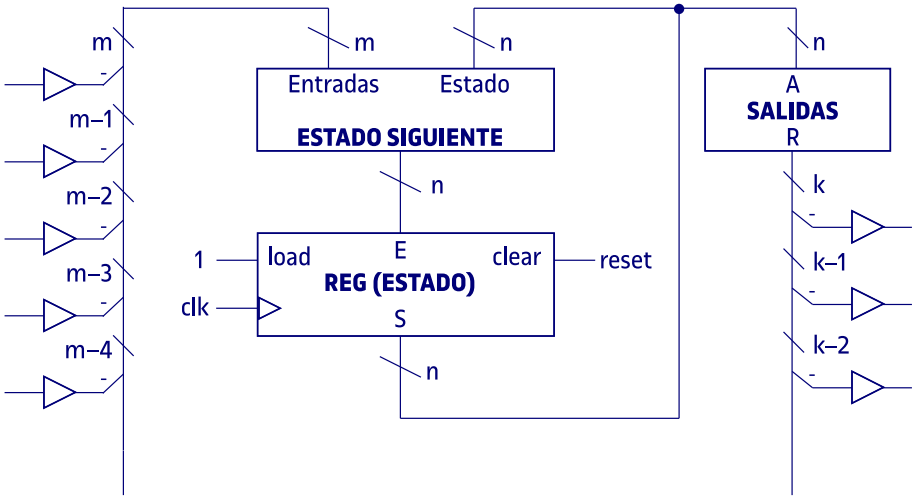
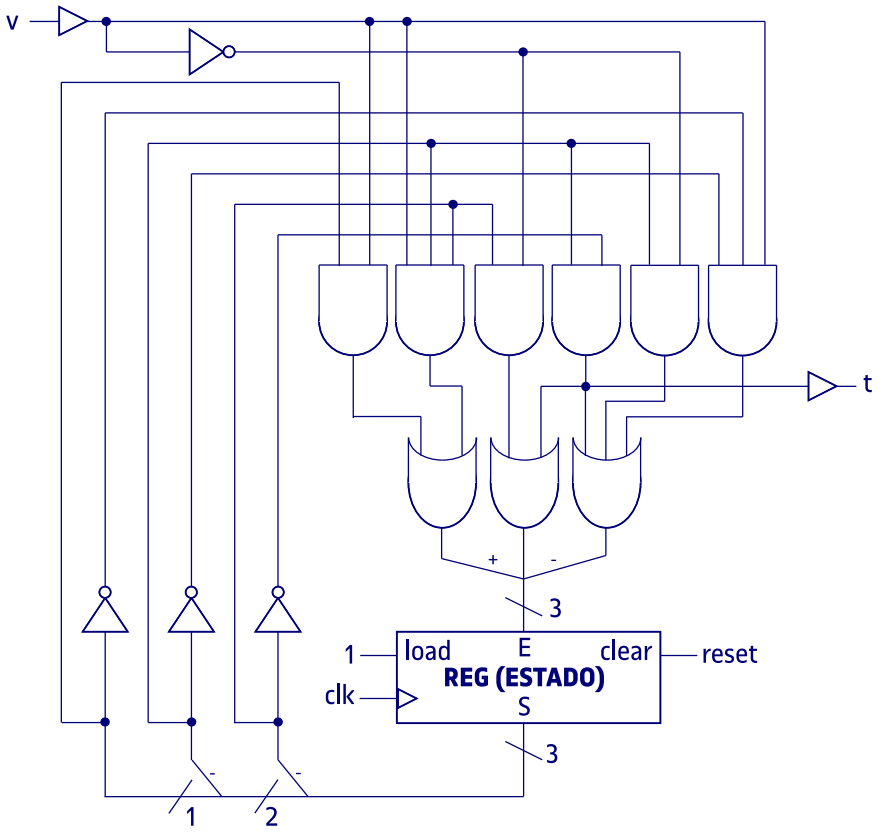


Figura 6. Esquema de la FSM controladora del timbre de aviso de entrada



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 corresponde a las funciones lógicas de las tablas de transición y de salida del controlador).

En casos más complejos, es conveniente usar bloques combinacionales más grandes como los multiplexores o los descodificadores, y también memorias ROM.

En este material, se mostrará la implementación con puertas lógicas de la mayoría de los circuitos, tanto a partir de las tablas de transición y de salidas como con otro método, que se explicará en su momento.

En cuanto a las implementaciones con bloques combinacionales o ROM, se dejan como ejercicio, aunque, como se ha visto, se pueden obtener directamente de las tablas de transición y de salidas.

Actividades

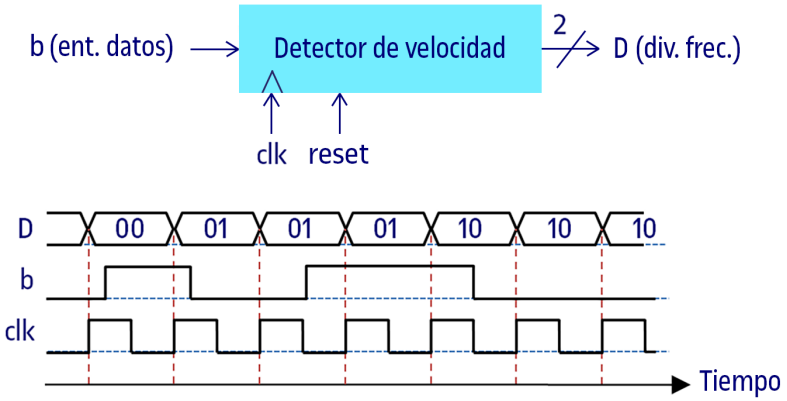
Actividad 1

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 la máquina de estados de 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 a que recibe los pulsos a 1. Los valores de salida pueden ser: 00 (no hay detección), 01 (ninguna división, el pulso en 1 dura lo mismo que un pulso de reloj), 10 (el pulso dura

dos ciclos de reloj) y 11 (el pulso de entrada está en 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 encontraréis una ilustración del funcionamiento del detector de velocidad.

Figura 7. Esquema del detector de velocidad de transmisión



Observad que, en el segundo pulso a 1 de *b*, la salida pasa de 00 a 10 progresivamente. Es decir, cuenta el número de ciclos en que se detecta la entrada en 1. Además, permanece en el mismo estado cada vez que finaliza una cuenta, puesto que, justo después de un ciclo con la entrada a cero, el circuito mantiene la salida en la última cuenta que ha hecho.

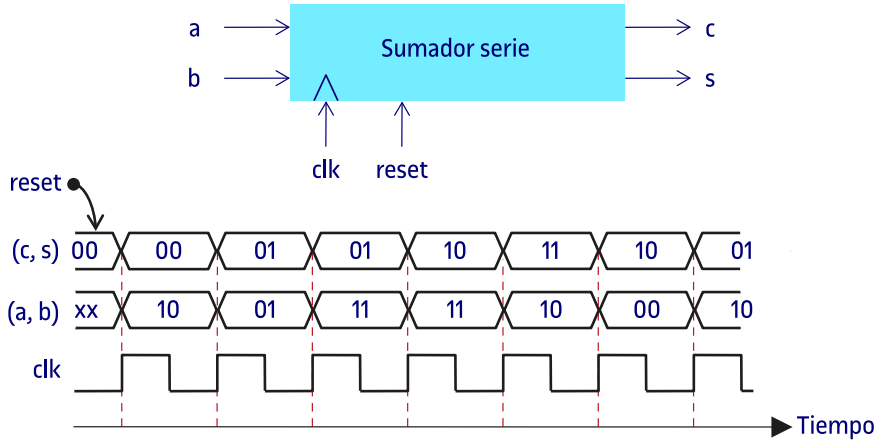
Haced el diagrama de estados y la tabla de transiciones correspondiente.

Actividad 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 tiene que 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



Haced primero 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.

1.3. 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 un ancho de unos cuantos bits.

En estos casos, hay 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).

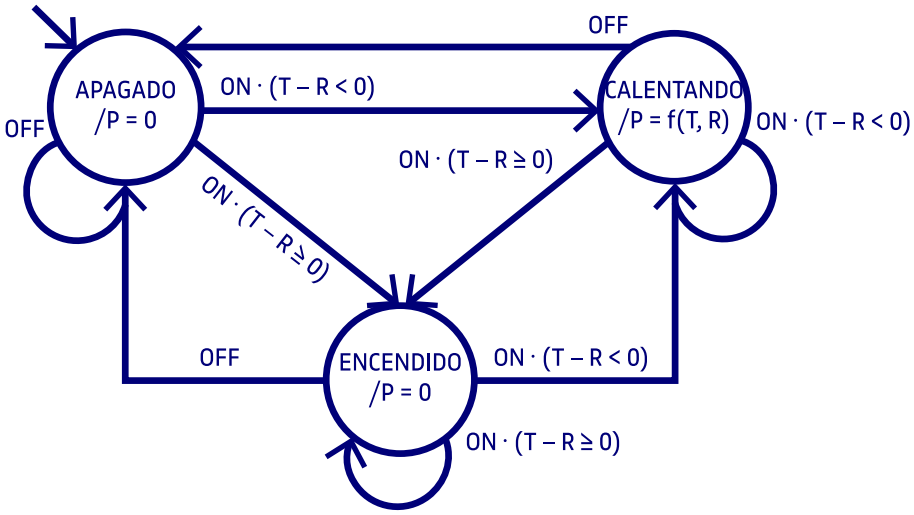
Para ilustrarlo, se puede pensar en un calefactor con un termostato que lo controle: en función de la diferencia entre la temperatura deseada (la referencia) y la que se mide (el estado del sistema que se controla), se determina la potencia del calefactor (el estado siguiente del controlador). En este 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 tiene que 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 desea.

El grafo de estados de la figura 9 es el correspondiente a un modelo EFSM del sistema que se ha comentado. En el grafo, ON/OFF' es una única señal de entrada de un bit que se representa como ON cuando es a 1 y como OFF cuando es a 0; 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, $f(T, R)$.

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.

Figura 9. Grafo de estados de un termostato



Hay que tener presente que los cálculos de las condiciones como por ejemplo ON·($T - R < 0$) u ON·($T - R \geq 0$) tienen que dar siempre un valor lógico. (En ambos casos, hay un resto que se compara con 0, dando como resultado un valor lógico que se puede operar, haciendo el producto lógico, con ON). Los cálculos para los valores de salida como $f(T, R)$ no tienen esta restricción, puesto que pueden ser de más de un bit de salida, como es el caso de P, que usa 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. Tal 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, puesto que es la base de los temporizadores, relojes, velocímetros y otros componentes útiles para los controladores.

En este caso, es un contador que se diseñará de manera 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 llegue una señal de inicio (*begin*) y se parará 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* tiene que ser 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 «Circuitos lógicos secuenciales» 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 al 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 hacer 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, tiene que 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 que M y C fueran de 8 bits tendría, como mínimo, $2^8 = 256$ estados.

Las EFSM permiten representar el comportamiento de un contador de manera más compacta porque los cálculos con C se pueden dejar como operaciones con datos almacenados con 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 ahí su nombre) de manera parecida a como se cambia de estado. Del mismo modo, la materialización del circuito correspondiente necesitará un registro para cada variable, además del registro para el estado.

Las variables son análogas en los estados: pueden cambiar de valor en cada transición de la máquina. Así pues, así 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 usa 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^+).

Variables y estados

En general, cualquier variable de una EFSM se puede transformar en información de estado y generar una máquina de estados con un número de estados que puede llegar a ser tan alto 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, puesto que «tiene que recordar» cuál es el valor de la cuenta actual (en este caso, el producto es por 1, puesto que la EFSM original solo tiene un estado).

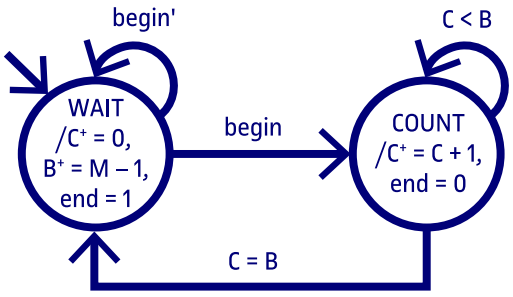
Desde otra perspectiva, se puede ver una variable como un elemento interno que, por un lado, proporciona un dato de entrada (en el contador, sería C) y, por otro, recibe un dato de salida que será la entrada en el ciclo siguiente (en el contador, sería C^+). En otras palabras, el contenido actual de una variable (en el contador, C) forma parte de las entradas de la EFSM y, entre las salidas de la misma máquina, hay la señal que transporta el contenido de la misma variable en el estado siguiente (en el contador, C^+).

De este modo, se puede hacer un modelo de EFSM para un contador autónomo que tenga un único estado en 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 autónomo, es decir, tiene dos. Uno de espera (WAIT) de 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.

El contador programable tiene dos salidas: la del mismo valor de la cuenta, que se almacena en una variable C, y otra de un bit, *end*, que indica cuando está contando o cuando está en 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 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 tiene que transcurrir, como mínimo, un ciclo de reloj.

Tal como ya se ha visto en el ejemplo del calefactor con termostato, las entradas y las salidas no se representan en binario sino con expresiones simbólicas, es decir, con símbolos que representan operadores y operandos. Para las transiciones se usan 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:

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

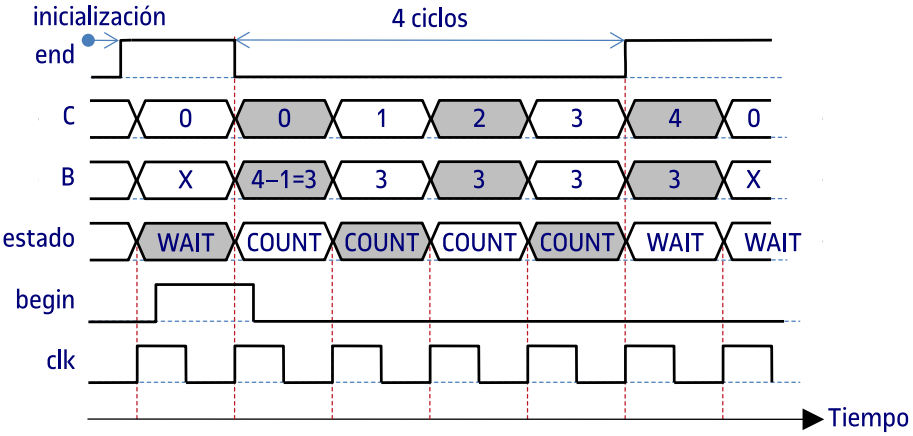
- Para las variables, se expresa el valor que adquirirán en el estado siguiente. Por ejemplo, en el estado COUNT, la variable C se tiene que incrementar en una unidad para el estado siguiente. Para que quede claro, se pone el superíndice con el signo de más: $C^+ = C + 1$.

Nota

Es muy importante tener presente que el contenido de las variables cambia al acabar el estado actual.

Para que quede más claro el funcionamiento de este contador programable se muestra, a continuación, en la figura 11, un diagrama de tiempo para una cuenta de hasta 4. Se trata de un caso que se inicia después de un *reset*, por lo cual *end* es 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 B y C será 4 – 1 y 0 respectivamente. Por lo tanto, al llegar a COUNT, $B = 3$ y $C = 0$.

Figura 11. Cronograma de ejemplo para el contador



Hay que fijarse en que el número de ciclos que transcurren entre el ciclo de reloj en 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. Esto se podría evitar haciendo que C^+ se calculara con módulo *M*, de forma que, en lugar de tomar el valor 4, se tomara 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.

Con vistas 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 hará referencia con los términos entre paréntesis, tal como aparecen en la tabla anterior.

Expresiones lógicas

Las expresiones de resultado lógico también se pueden identificar con señales de un único bit que, a su vez, se tienen que 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. Sin embargo, lo más habitual es que una determinada señal de salida pueda tener varias 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 varios bits que selecciona cuál de los resultados se tiene que asignar a una determinada salida. Así pues, haría falta un selector para C^+ y otro para B^+, los dos de un bit, puesto 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	Almacenamiento del valor constando 0
$C^+ = C + 1$	1	Almacenamiento del resultado del incremento

Hay que tener en cuenta que los selectores se materializan, habitualmente, en 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 al hecho de 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, hay que tener en cuenta que las condiciones ($C < B$) y ($C = B$) son opuestas y, por lo tanto, es suficiente, por ejemplo, usar ($C < B$) 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

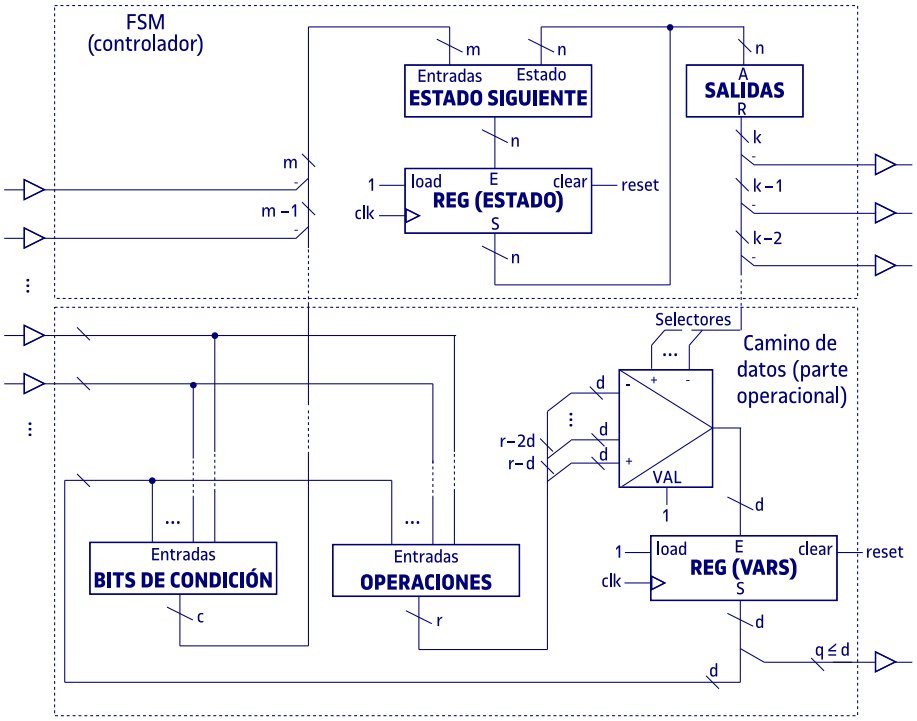
Antes de pasar a la implementación de las funciones de las tablas de verdad anteriores, hay que ver cómo se construye una de estas EFSM. En general, la arquitectura de este tipo de circuitos secuenciales separa la parte que se ocupa de hacer las operaciones con los datos numéricos de la parte que trata 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 al 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 también selectores para los resultados a almacenar en los registros internos. La segunda parte es la que hace las operaciones con datos o camino de datos, que 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 de salida y las que correspondan 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.

Camino de datos

La denominación de *camino de datos* de la parte operacional procede del hecho de que los datos la atraviesan siguiendo un camino u otro según la operación que se haga.

Figura 12. Arquitectura de una EFSM

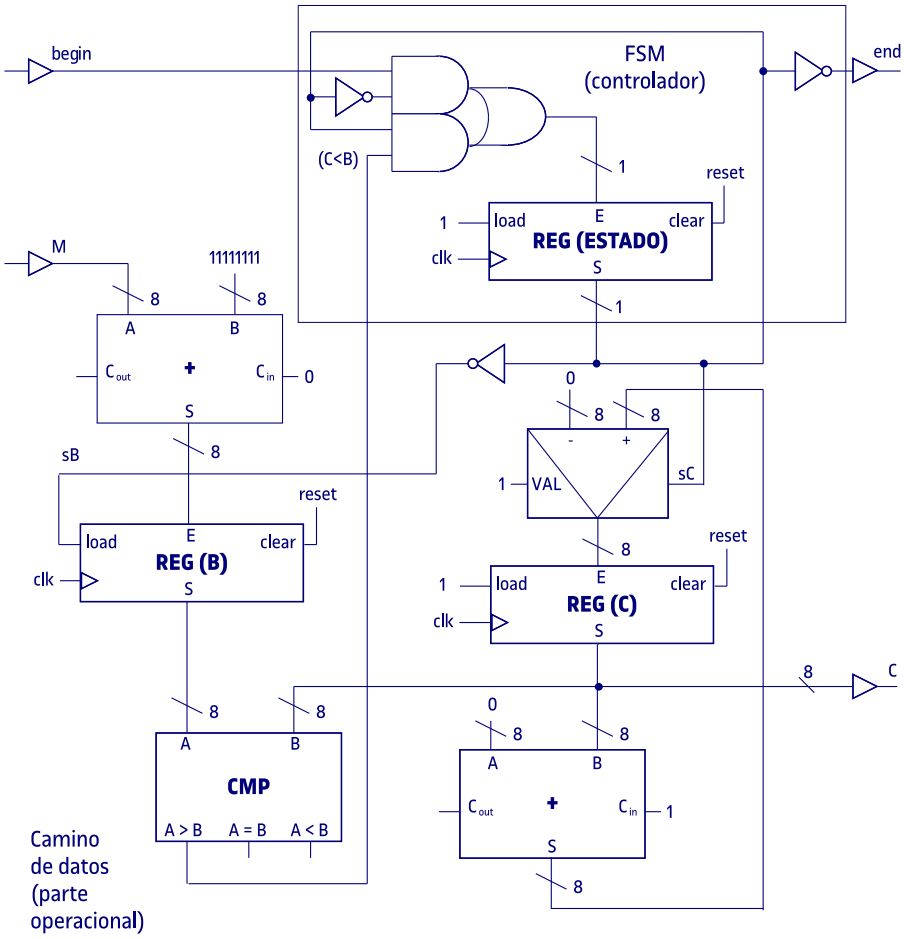


En resumen, pues, la EFSM tiene dos tipos de entradas (las señales de control de un bit y las señales de datos, de múltiples bits) y dos tipos de salidas (señales de un único bit, que son de control para elementos exteriores, y señales de datos resultantes de alguna operación con otros datos, internas o externas). El modelo de construcción que se ha mostrado las organiza de forma 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 hay, 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 construir el contador hay que seguir el mismo modelo. En la figura 13 se puede ver el circuito completo. La parte superior se corresponde con la unidad de control, que calcula la función de excitación, que es $s^+ = s' \cdot begin + s \cdot (C < B)$, y las de salida (*end*, hacia el exterior, y *sB* y *sC*, que son selectores), que se obtienen directamente del estado o del estado complementado. Se ha mantenido el registro de estado, como en el esquema anterior, pero es un registro de un único bit que se puede implementar con un 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 un lado, el cambio de valor y, por otro, el mantenimiento del contenido.

Figura 13. Circuito secuencial correspondiente al contador del ejemplo



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 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. Dado que 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.

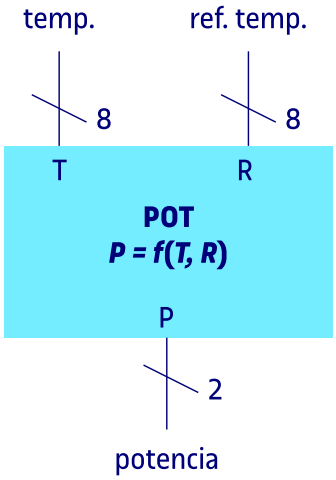
Los contadores programables son la base de una serie de módulos habituales, como los temporizadores. Un **temporizador** es un contador programable, con una entrada M para indicarle cuántos ciclos de reloj tiene que estar contando. Si se sabe el periodo del reloj, M determina el tiempo que tiene que transcurrir hasta que no active la señal de finalización.

Actividades

Actividad 3

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

Figura 14. Esquema de entradas y salidas del módulo POT



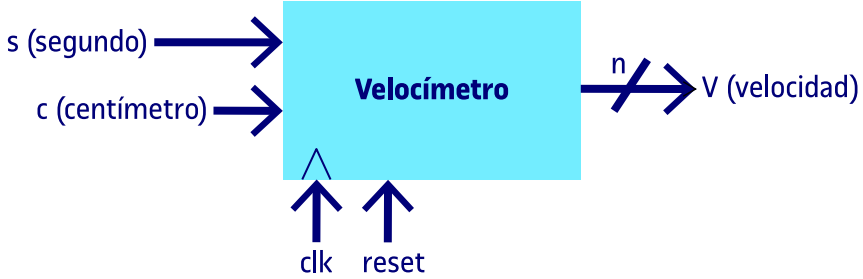
Actividad 4

En muchos casos, de los contadores solo interesa la señal de final de cuenta. En este sentido, no hace falta 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 a la inversa. De este modo, basta con un único registro cuyo valor se vaya decrementando hasta 0. Modificad la EFSM del contador del ejemplo para que se comporte de este modo y diseñad el circuito resultante.

Actividad 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 a cada segundo, y una señal c, que es 1 en el periodo de reloj en que se ha recorrido un centímetro, y, como salida, el valor de la velocidad en cm/s, V, que se tiene que actualizar a cada segundo (figura 15). Hay que tener presente que s y c pueden pasar en el mismo instante. El valor de la velocidad se tiene que mantener durante todo un segundo. Al arrancar, durante el primer segundo tendrá que ser 0.

Figura 15. Esquema de entradas y salidas del módulo del velocímetro

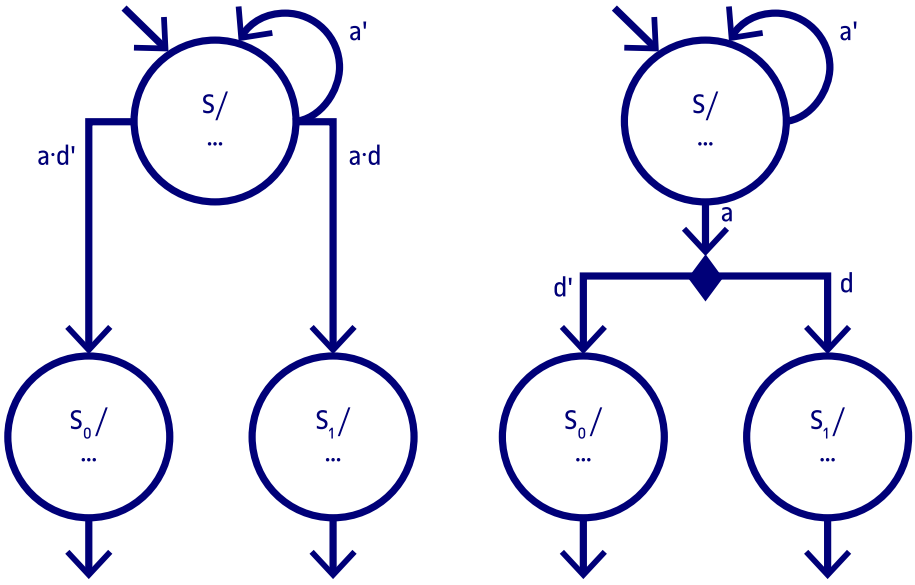


2. Máquinas de estados algorítmicas

Las representaciones de las máquinas de estados extendidas permiten vincular cada estado a un conjunto de operaciones concreto y, además, indicar qué condiciones se deben cumplir para hacer una transición u otra desde el estado actual. A menudo, las expresiones lógicas de estas condiciones son complejas.

Viéndolo en un ejemplo simple (figura 16), en el caso de un estado S que espere que una señal *a* se ponga en 1 y, en el momento en que lo haga, pase a un nuevo estado S_0 o S_1 según otra señal de entrada *d*, los arcos de salida de S hacia estos estados tendrán las expresiones $a{\cdot}d'$ y $a{\cdot}d$, respectivamente. Esta situación es muy habitual: parte de las expresiones lógicas de las transiciones se repite y hace más compleja la representación del grafo correspondiente. Una solución es crear bifurcaciones de los arcos, de forma que cada rama tenga asociado un término de la expresión original: en el ejemplo, habría un arco de salida etiquetado con la expresión *a* del que saldrían dos ramas, *d'* y *d*, que irían a parar a S_0 y S_1. Así pues, la máquina de estados tendría operaciones vinculadas a los estados, y bifurcaciones vinculadas a las transiciones.

Figura 16. Grafos de EFSM convencional (izquierda) y con bifurcaciones (derecha)



Una máquina de estado algorítmica o ASM (del inglés, *algorithmic state machine*) es una máquina de estados con transiciones que pueden formar ramas. Es decir, en la que una transición puede ser final, si lleva a un estado, o condicional, si lleva a una bifurcación. Se llama algorítmica porque puede actuar de modelo de comportamiento de un algoritmo: cada paso del algoritmo es un

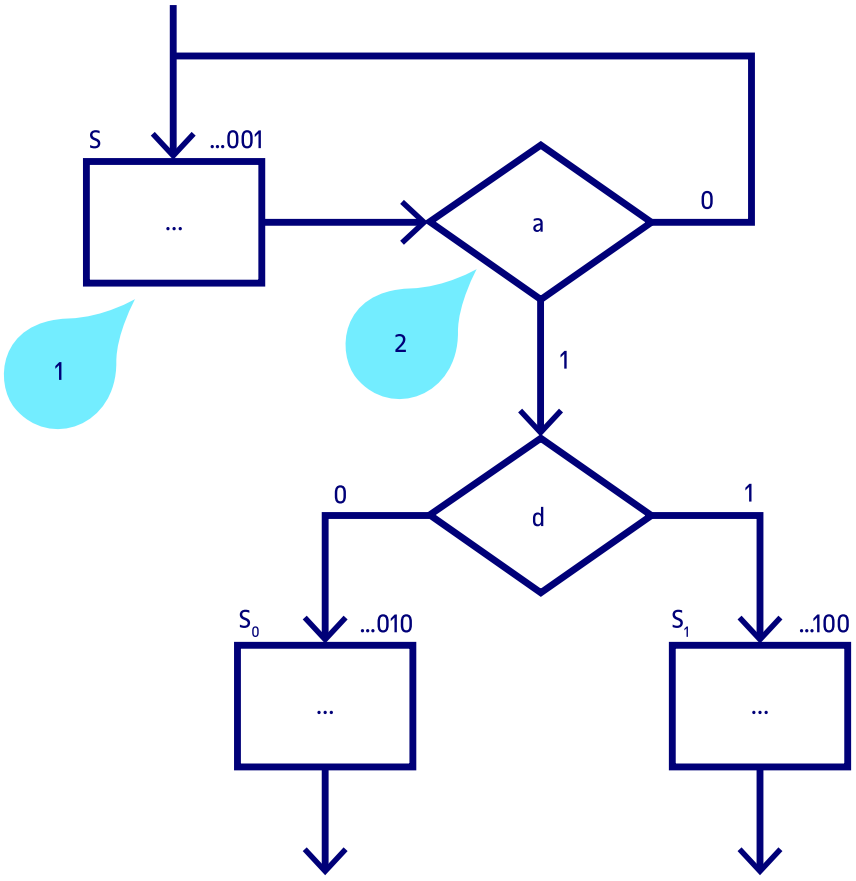
estado, y la secuencia de estados que se recorra no tiene por qué incluir todos los estados de la máquina (ejecución condicional) y puede incluir repeticiones (ejecución iterativa).

En este bloque, se verá cómo hacer circuitos a partir de las ASM que describan el comportamiento de algoritmos de control y de cálculo, también.

2.1. Representación de máquinas de estados algorítmicas

Habitualmente, las ASM se representan usando grafos que contienen dos tipos de nodos: los de procesamiento (1) y los de decisión (2), tal como se ve en la figura 17.

Figura 17. Representación de una máquina de estados algorítmica



Los nodos de procesamiento, en forma de caja rectangular, son los estados de la máquina, y los de decisión, en forma de rombo, representan las bifurcaciones de los arcos de transición.

Los nodos de decisión tienen dos salidas, una para cuando la expresión que contienen es cierta (1) y la otra para la situación contraria (0).

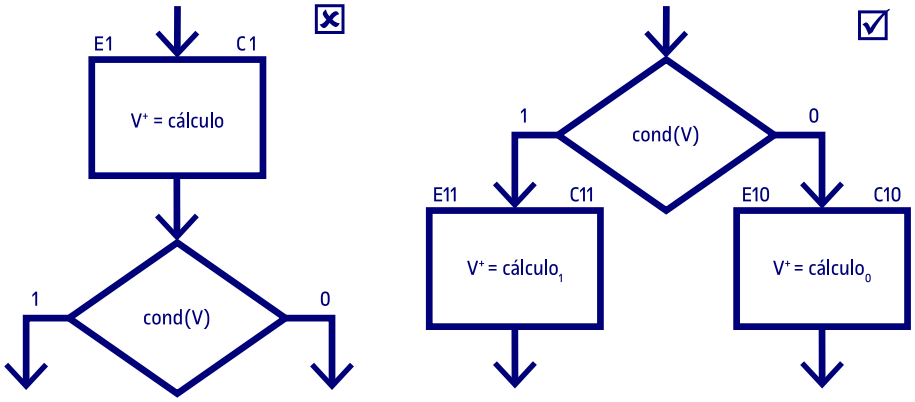
El nombre del estado y su código binario se sitúan fuera de las cajas rectangulares, y dentro hay las operaciones que se llevan a cabo en cada estado durante un ciclo de reloj.

Los diagramas que representan las ASM son muy similares a los diagramas de flujo (*flowchart*, en inglés) porque usan rectángulos y rombos con un significado muy similar: en unos hay cálculos para determinar los nuevos valores de las variables y de las salidas, y en los otros, cálculos para determinar si las expresiones lógicas son falsas o ciertas.

La diferencia es que, en un diagrama de una ASM, los rectángulos representan estados de la máquina. Esto quiere decir que los cálculos que hay tanto en los mismos rectángulos como en los rombos que se encuentran entre los arcos de salida y los rectángulos de llegada se hacen con el contenido actual de las variables, con independencia de cómo aparezcan en el diagrama.

En los casos en que las decisiones dependan de variables que se modifiquen en el mismo estado, como se muestra a la izquierda de la figura siguiente, hay que tener en cuenta que las expresiones de las decisiones usan el valor actual aunque aparezcan en un nodo diferente. Para evitar confusiones, es recomendable que, en estas situaciones, los diagramas de los ASM pongan primero las cajas de decisión y después las de estado, tal como se ilustra en la parte derecha de la figura 18.

Figura 18. Formas desaconsejada y aconsejada de colocar estados con cálculos que afectan bits de condición



Con todo, las ASM se pueden usar tanto para diseñar controladores como módulos de procesamiento de datos.

2.2. Materialización de máquinas de estados algorítmicas

Los circuitos que materializan las ASM se obtienen del mismo modo que con las EFSM: a partir de las tablas de transición y de salidas se pueden obtener las expresiones lógicas a sintetizar o se pueden usar directamente para determinar el contenido de las memorias ROM que generen las funciones correspondientes.

De manera alternativa, las funciones de transición se pueden obtener directamente del diagrama de las ASM, tal como se verá a continuación. La única cuestión es que la codificación de los estados será de tipo *one-hot bit*. Con esta codificación, cada estado tiene asociado un biestable que se pone a 1 cuando la máquina de estados se encuentra, precisamente, en aquel estado. Hay que tener en cuenta que, en máquinas que usan muchas variables, el número de biestables que se usará para la codificación de los estados será relativamente pequeño en comparación con la cantidad de registros de trabajo del circuito global.

Con este tipo de unidades de control, el bit a 1 que señala en qué estado está la máquina correspondiente va desplazándose entre los diversos biestables que lo almacenan según la transición que se active. Para ello, hay una correspondencia entre nodos de procesamiento y biestables del circuito correspondiente, y entre nodos de decisión y demultiplexores.

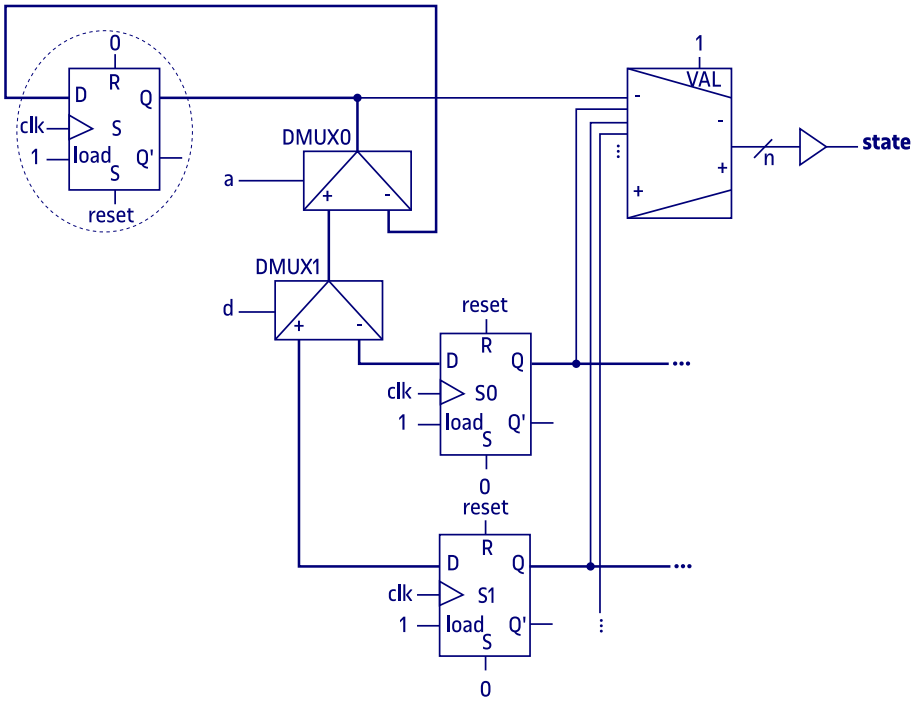
En la figura 19 hay el circuito de la unidad de control del diagrama de flujo que se ha visto en la figura 17. En el esquema, los arcos están marcados con buses de línea gruesa para facilitar la comparación entre circuito y diagrama. Además, los nodos de estado se identifican con el mismo nombre con que aparecen en el diagrama de flujo correspondiente. Si estos nodos tuvieran más de una entrada, se tendría que poner una puerta OR que las una.

Los nodos de decisión son elementos que hacen lo mismo que uno de multiplexor: transmiten la entrada a la salida seleccionada. En la figura hay dos: DMUX0 y DMUX1. Por ejemplo, DMUX0 «hace pasar» el 1 que viene de S hacia S otra vez o hacia el otro nodo de decisión según la condición *a*.

El biestable S, que se corresponde con el estado inicial y que está rodeado en la figura, se tiene que poner a 1 cuando se hace un *reset* del circuito. Para ello, la señal de *reset* está conectada a la entrada *set* del biestable.

En caso de que haga falta el código del estado, hay un codificador binario para transformar la codificación de un bit activo en numérica, marcada en la figura como salida *state*.

Figura 19. Ejemplo de unidad de control de una máquina de estados algorítmica



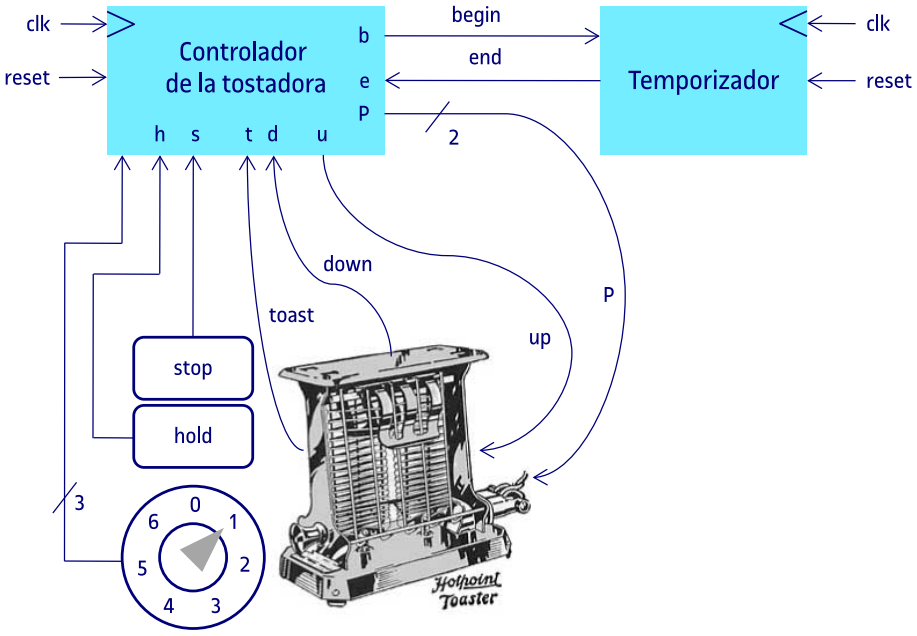
Hay que tener presente que, para completar la unidad de control, hay que incorporar el circuito que genera las salidas. Esto se debe hacer con los métodos que ya se han visto, generando expresiones lógicas a partir de las tablas de salida o usándolas para obtener el contenido de las memorias ROM correspondientes.

2.3. Implementación de controladores con máquinas de estados algorítmicas

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 no se pulse el botón anterior. Además, tiene una rueda que permite seleccionar seis grados diferentes de tueste y que también puede estar apagada (posición 0). Para ponerla en marcha solo hay que 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 tueste durará tantos periodos de tiempo como número de posición tenga el selector.

El esquema de la tostadora se muestra en la figura 20 (la tostadora representada es un modelo de principios del siglo XX que, evidentemente, tenía un funcionamiento muy diferente del presentado). En la figura también hay el nombre de las señales para el controlador. Hay que tener presente que la posición de la rueda se puede leer a través de la señal *R* de tres bits (los valores posibles están en el rango [0, 6]) y que la potencia *P* de los elementos calefactores se codifica de forma 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 suficiente calor 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 que se hará la tostada. A más valor, más lapsos de tiempo y más tostada quedará la rebanada de pan. Estos periodos de tiempo se controlarán con un temporizador, mientras que el número de periodos se contará internamente con la ASM correspondiente.

Figura 20. 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 ASM. En la figura 21 se puede observar el diagrama completo.

Es recomendable que las acciones y las condiciones que se describen en los nodos del diagrama de estados sean tan generales como sea posible. De este modo, son más comprensibles y no quedan vinculadas a una implementación concreta.

A modo de ejemplo, en los nodos de decisión solo hay la indicación textual de las condiciones que se deben cumplir. Por comparación, en los 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 20. Se puede comprobar que, en

un caso algo más complejo, las acciones serían más difíciles de interpretar. En cambio, las condiciones resultan más comprensibles y, además, se pueden trasladar fácilmente hacia expresiones que, a su vez, se pueden materializar de manera bastante directa.

El estado inicial de la ASM es IDLE, en que los radiadores de la tostadora están apagados (P = 00), no se hace ninguna acción (*up* o *begin*), ni se hace saltar la rebanada de pan (u = 0), ni se pone en marcha el temporizador (b = 0) y, además, se pone a cero la variable C, lo cual tendrá efecto para el estado siguiente, tal como se indica con el símbolo más en posición de superíndice.

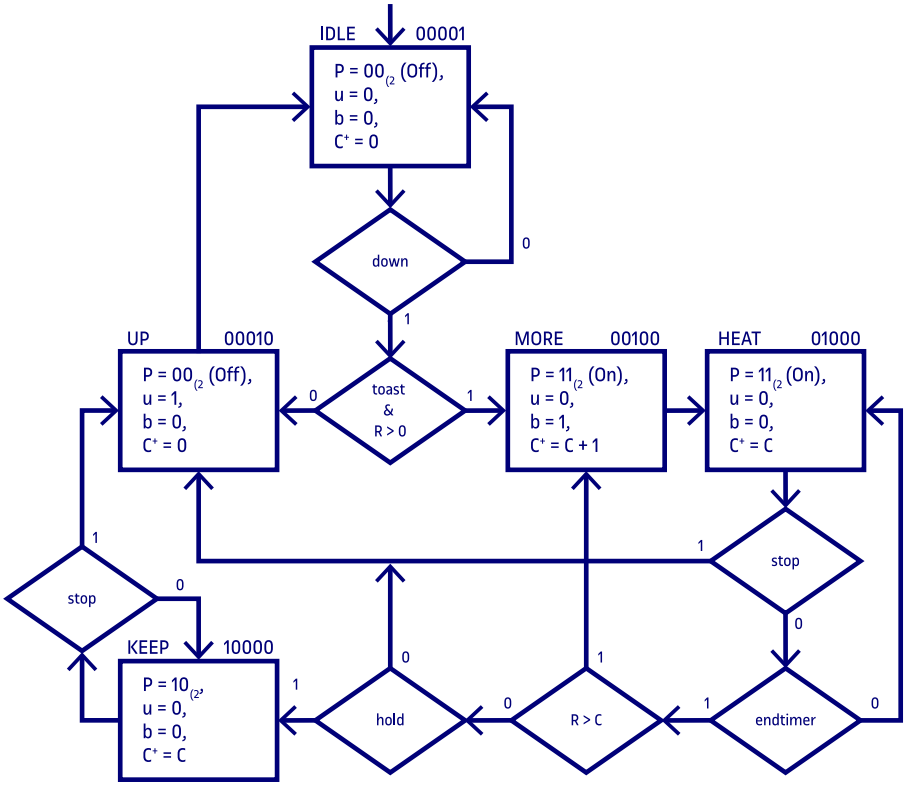
Nota

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 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 el nodo de estado cuando se pasa al estado siguiente.

En el estado IDLE se efectúa la espera de que se ponga una rebanada de pan en la tostadora. De ahí que la condición que se tiene que cumplir en el arco de salida 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 la de 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 apoyo del pan (o bien no se había metido nada o bien había una rebanada pero la rueda estaba en la posición 0) y se devolverá, finalmente, al estado inicial de espera. En el segundo caso, se pondrán en marcha los calefactores para iniciar el tueste. También se pondrá el contador interno de intervalos de tiempos a 1 y se activará el temporizador para que avise del momento en que se acabe el periodo de tiempo del intervalo actual.

Es interesante observar cómo, en las ASM, los nodos de estado solo tienen un arco de salida que se va dividiendo a medida que atraviesa los nodos de condición. Por ejemplo, el arco de salida de IDLE tiene tres destinos posibles según las condiciones que se cumplan: IDLE, MORE y UP. De manera muy diferente a cómo se tendría que hacer en el modelo de EFSM, en que un estado similar debería tener tres arcos de salida con expresiones de condición completas para cada arco.

Figura 21. Diagrama de una ASM para el controlador de una tostadora



Tal como se ha comentado, el número de intervalos de tiempo que deben transcurrir para hacer la tostada está determinado por la intensidad del tueste, que se gradúa mediante la posición de la rueda de control, que puede variar de 1 a 6. En caso de que se ponga una rebanada de pan y la posición sea 0, es decir, si $R = 0$, se hace subir de nuevo, aunque no sea la opción habitual. En el estado MORE se incrementa un contador interno, C, que tiene la función de contar los intervalos de tiempo que hay en funcionamiento. Tened en cuenta que se ha seguido la recomendación ejemplificada en la figura 18: la condición ($R > C$) en función de una variable precede un estado que la modifica (MORE).

De este estado se pasa al de tostar (HEAT), en que se queda hasta que se pulse el botón de parada (*stop*) o se acabe el periodo de tiempo en curso (*endtimer*). En el primer caso, pasará a UP para hacer subir la tostada tal como esté en ese momento y volverá, finalmente, a IDLE. En el segundo caso, comprobará cuántos periodos tienen que pasar todavía hasta conseguir la intensidad deseada de tueste. 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 el tueste ($R > C$) hay que hacer subir la tostada (ir a UP), salvo que el botón de mantenimiento (*hold*) esté pulsado. En este caso, se pasa a un estado de espera (KEEP) que deja los elementos calefactores en una potencia intermedia que mantenga la tostada caliente. Al pulsar el botón de parada, se pasa a UP para hacer saltar la tostada y apagar la tostadora.

Para la materialización de la ASM hay que representar las expresiones de todos los nodos con fórmulas lógicas. Por este motivo conviene hacer lo mismo que se ha hecho con los 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*·(*R* > *C*)
stop	*s*
endtimer	*e*
R > *C*	(*R* > *C*)
hold	*h*

En la tabla anterior, la condición ($R > 0$) se ha sustituido por la expresión ($R > C$) porque en IDLE, la variable C siempre valdrá 0, bien porque se ha inicializado la máquina o bien porque procede de UP o del mismo IDLE, puesto que los dos hacen $C^+ = 0$.

Para construir el circuito secuencial de la tostadora, que se corresponde con la ASM de la figura 21, haría falta, primero, obtener la parte correspondiente a las funciones de transición o de cálculo del nuevo estado. Como las tablas serían muy grandes, es complejo generar las expresiones de estas funciones y la memoria ROM correspondiente tendría una medida grande. Por lo tanto, se hará con el método que se ha explicado en el apartado «Materialización de máquinas de estados algorítmicas».

La parte operacional solo tiene que materializar los pocos cálculos que hay, que son:

- Para el registro de conteo C, $C^+ = 0$ y $C^+ = C + 1$, que se seleccionarán con la señal sC = 00, 01, 10, respectivamente.
- La comparación con la referencia R, $R > C$, que da el bit de condición r.

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

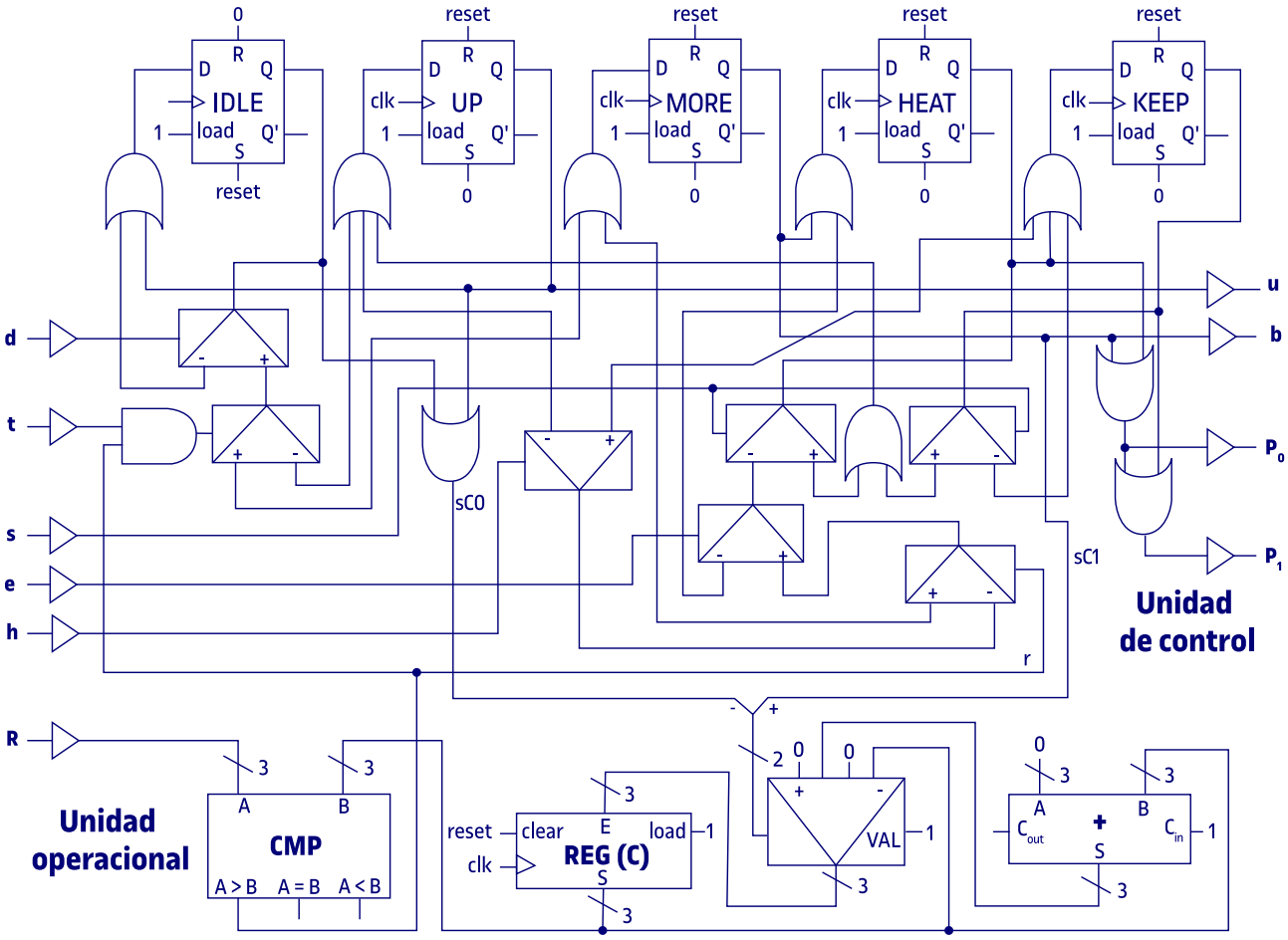
- sC_0 = IDLE + UP
- sC_1 = MORE
- u = UP
- b = MORE
- p_0 = MORE + HEAT

- p_1 = MORE + HEAT + KEEP

Así pues, como la codificación de los estados es de un bit activo a la vez, basta con tres puertas OR para generar sC_0, p_0 y p_1.

Con todo, el esquema del controlador de la tostadora es el que se muestra a continuación, en la figura 22.

Figura 22. Esquema del circuito del controlador de una tostadora



A pesar de que se puede optar por hacer un diagrama del circuito de la unidad de control que sea similar al del grafo de la ASM, aquí se ha hecho uno de más compacto.

Actividades

Actividad 6

Construid la tabla de verdad de las funciones de transición del controlador de la tostadora.

Actividad 7

Modificad la ASM de forma que, cuando se ponga 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 pulse *stop*.

2.4. Implementación de módulos de procesamiento de datos con máquinas de estados algorítmicas

Además de controladores con muchas señales de entrada y salida, los diagramas de las ASM pueden representar muy bien todo tipo de algoritmos de procesamiento de datos, en que se trabaja, sobre todo, con variables.

Como ejemplo de esto, se construirá un multiplicador secuencial. La idea es reproducir el mismo procedimiento que se hace con las multiplicaciones a mano: se hace un producto del multiplicando para 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, puesto 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 en una suma de multiplicandos decalados según la posición de cada 1 del multiplicador.

Para verlo más claro, a continuación se hace el producto de 1011 (multiplicando) por 1010 (multiplicador):

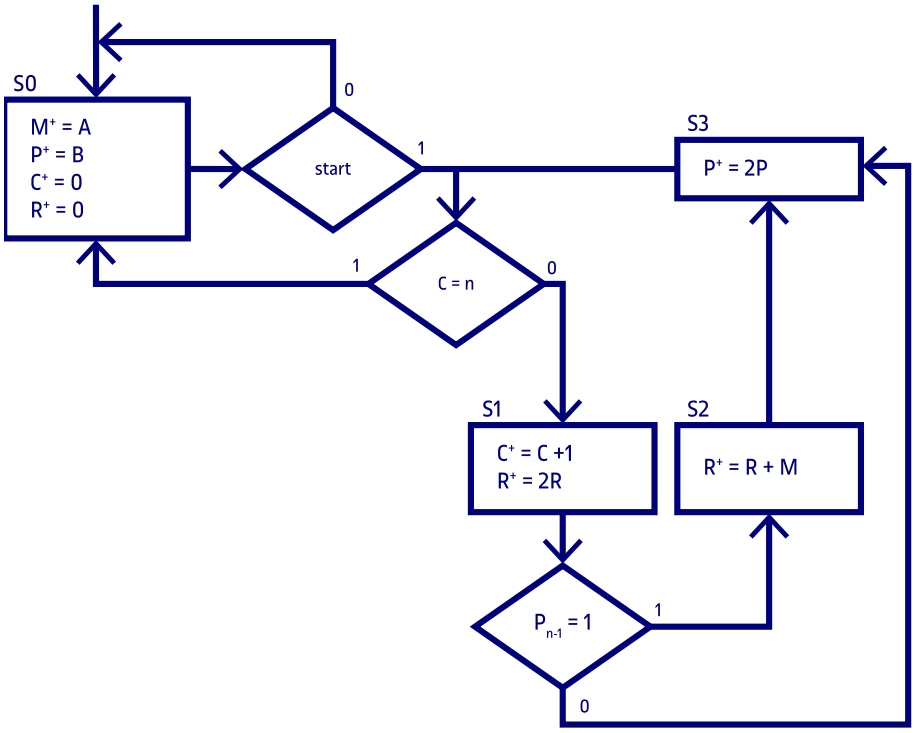
```
          1  0  1  1
        x 1  0  1  0
    ------------------
          0  0  0  0
       1  0  1  1
    0  0  0  0
 1  0  1  1
-------------------------
 0  1  1  0  1  1  1  0
=========================
```

Para materializar un circuito que haga la multiplicación con este procedimiento hace falta, primero, diseñar el diagrama de la ASM con las operaciones y las decisiones que se toman.

El estado inicial S_0 también será el final, con una espera a que una señal de inicio *start* se ponga a 1.

En la figura 23 hay 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 que toca hacer un determinado producto se obtiene de un contador, C. El contador sirve, además, de control del número de iteraciones del algoritmo, que es el número de bits de los operandos A y B (n).

Figura 23. ASM de un multiplicador binario



Justo es decir que el resultado R tendrá que tener un ancho de $2n$ bits y que el contador tiene suficiente 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, con m siendo lo bastante grande como para representar cualquier número más pequeño que n.

El algoritmo de multiplicación tiene que empezar (S_0) cargando el multiplicando a M y el multiplicador a 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 en R habrá el resultado.

A 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 pues, antes de sumar a R un nuevo producto, hay que multiplicarlo por dos, porque el resultado parcial que contiene es de la suma parcial previa, que se corresponde a los productos de M por los bits más significativos de P.

El proceso en S_2 es, simplemente, hacer la suma del producto de un bit en 1 de P por M con el resultado parcial previo a R. Con independencia de este proceso, en S_3 se hace un desplazamiento a la izquierda de P para obtener el multiplicador siguiente.

Los desplazamientos de B y de R ahorran el uso de multiplexores o de otro tipo de selectores, puesto que los bits con que se tiene que trabajar siempre están en posiciones fijas.

Para verlo más claro, a continuación hay 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 usa 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	

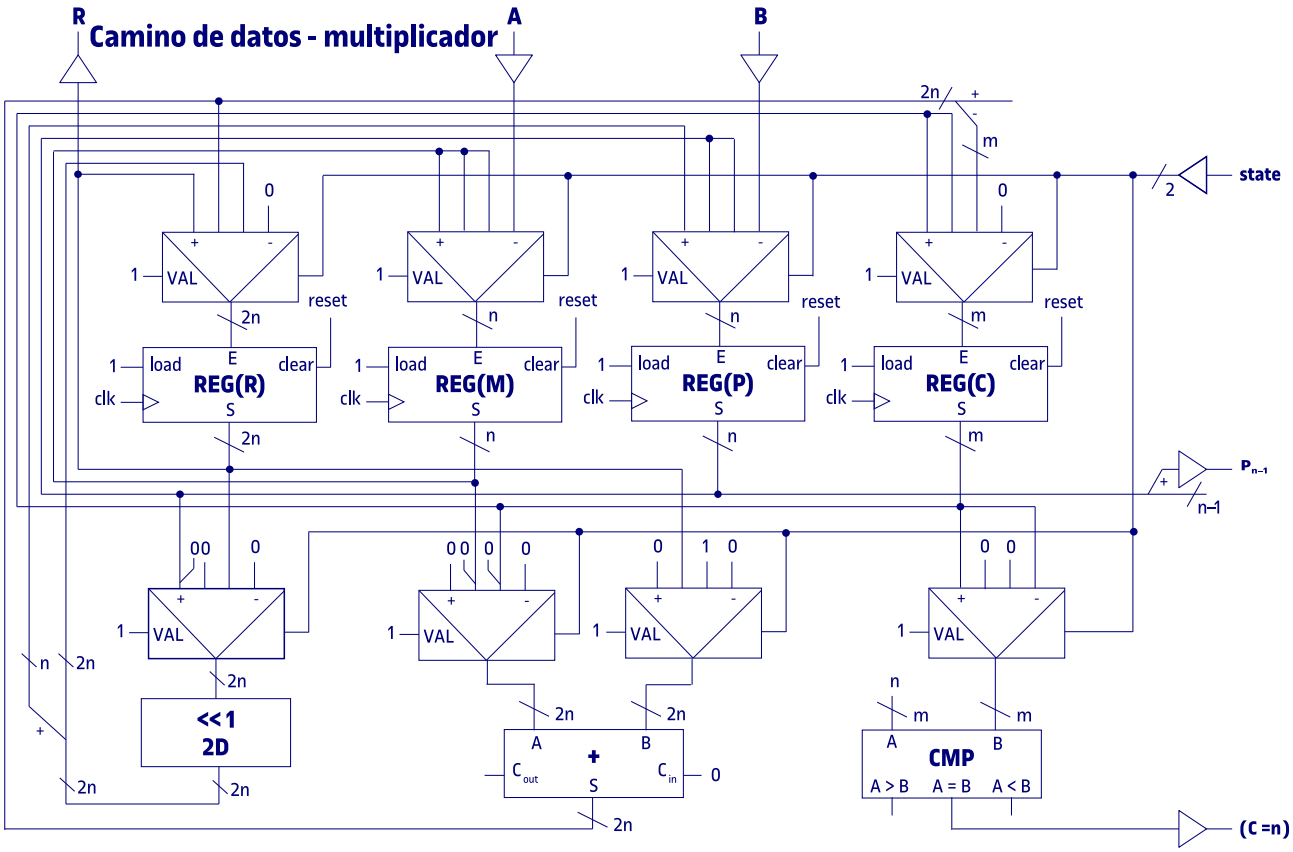
Para construir la unidad de procesamiento hace falta, primero, tener claros los recursos de memoria y de cálculo que debe tener. En cuanto a los primeros, basta con un registro para cada variable del algoritmo: M y P de n bits, R de $2n$ bits y C de m bits. En cuanto a los segundos, debe haber uno por operación diferente como mínimo. En este caso, tanto las sumas como los decaladores pueden ser compartidos porque se usan en estados diferentes, lo cual es positivo porque se reduce el número de recursos de cálculo. En este caso, solo hace falta un sumador para calcular $C + 1$ y $R + M$ porque son sumas que se hacen en estados diferentes (S_1 y S_2, respectivamente) y solo hace falta un decalador que se puede aprovechar para hacer el cálculo de $2R$ a S_1 y $2P$ a S_3.

El hecho de que el decalador y el sumador sean compartidos implica que tienen que 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 operandos a esta ancho: 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, a las entradas 1 y 2 del multiplexor

de la izquierda, se pasan a $2n$ bits añadiéndoles 0 por la izquierda, para no cambiar su valor. En la figura no se han puesto los anchos de los buses para mantener la legibilidad.

La unidad de procesamiento que se muestra en la figura 24 sigue una estructura muy regular para que sea fácil de leer: el dato que se carga en cada registro se selecciona con un multiplexor de buses en que cada entrada se corresponde con un número de estado y los recursos de cálculo reciben los datos del mismo modo. Con todo, el circuito final se puede simplificar en cuanto al uso de los multiplexores.

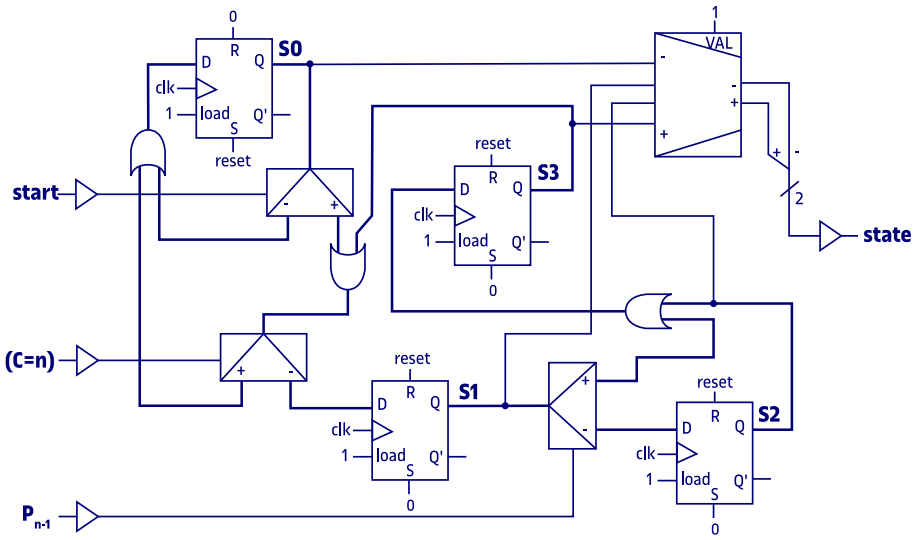
Figura 24. Unidad de procesamiento de un multiplicador en serie



La unidad de control se ha construido con las funciones de transición implementadas según el método que se ha comentado en el apartado 2.2. En la figura siguiente hay el esquema del circuito de control correspondiente al multiplicador. Las líneas de conexión de los arcos del diagrama de grafo correspondiente se destacan en negrita.

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.

Figura 25. Unidad de control de un multiplicador en serie



Actividades

Actividad 8

Rehaced los esquemas de los circuitos de la unidad de procesamiento y de la unidad de control, si es conveniente, para reducir al máximo el número de multiplexores y la cantidad de entradas de cada uno en la unidad de procesamiento.

Actividad 9

Diseñad la ASM de un módulo de cálculo para $P = A \cdot T^2/2 + V \cdot T + S$, en que A es la aceleración, V, la velocidad, S, el espacio inicial y T, el tiempo. El formato de todos los datos es el mismo (por ejemplo, números en complemento a 2 de 16 bits). El módulo tiene una señal de entrada *start* que se pone a 1 cuando hay que hacer un nuevo cálculo. Hay una restricción para la implementación porque solo se puede usar un multiplicador y, por lo tanto, en cada ciclo solo se puede hacer un producto.

En general, la materialización de una máquina de estados se hace partiendo de la representación que sea más adecuada. Las FSM que trabajan con señales binarias son más sencillas de implementar, pero es más difícil ver la relación con la parte operacional. Las EFSM solucionan este problema integrando cálculos en el modelo de sistema. Sin embargo, a veces resulta más adecuado simplificar la representación de las transiciones y verlo desde un punto de vista más algorítmico (ASM).

3. Arquitectura básica de un computador

Un computador es una máquina capaz de procesar información. Con esta definición, ciertamente, estamos en un mundo lleno de computadores, algunos de los cuales vemos como tales, como los portátiles y los de escritorio. Pero la gran mayoría no los percibimos como ordenadores porque se encuentran empotrados (y escondidos) 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 de estados algorítmicas especializadas, hasta los más complejos.

En este apartado se verá cómo están construidos los computadores y cómo funcionan, empezando el recorrido por la transformación de las ASM en computadores y acabando por las arquitecturas de los computadores más completos.

En primer lugar, se verán las ASM 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 donde se almacenan. De hecho, este tipo de máquinas se llaman *procesadores* porque «procesan» datos de acuerdo con un programa determinado.

A medida que se aumenta el repertorio de cosas que pueden hacer los programas, es decir, el conjunto de instrucciones diferentes que incluyen, las ASM que las pueden entender también se hacen 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 consigue que puedan ejecutar más instrucciones por unidad de tiempo. También se explicarán los diversos tipos de procesadores que puede haber según sus características.

Dado que los procesadores se ocupan de procesar información pero no de obtener ni de proporcionar, se tienen que acompañar de otros componentes que lleven a cabo estas funciones, es decir, que se ocupen de la entrada y salida de la información. El conjunto se denomina *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 de estados algorítmicas genéricas

Las ASM 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 por ejemplo el tiempo, el consumo de energía y el tamaño de los circuitos.

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.

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 hace falta que los sistemas resultantes tengan un coste razonable respecto al rendimiento que se obtiene. La batalla por la eficiencia está, pues, en el tiempo y en el consumo de energía: se quieren obtener resultados de forma cada vez más rápida y consumiendo poca energía.

Ejemplo de eficiencia

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. La eficiencia se podría medir en función del tiempo máximo de reproducción por carga completa de la batería.

Otra parte del coste tiene que ver con el desarrollo y posterior mantenimiento de los productos gobernados por ASM: 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 ASM nueva.

En el estado actual de la tecnología, lo que más pesa a la hora de materializar un producto es la funcionalidad (porque incrementa el valor añadido) y la facilidad de desarrollo (porque reduce el tiempo de puesta en el mercado) y mantenimiento posterior (porque facilita la actualización). De hecho, la tecnología permite ejecutar de manera eficiente gran parte de la funcionalidad. Solo unas pocas funciones son realmente críticas y hay que llevarlas a cabo con algoritmos específicos.

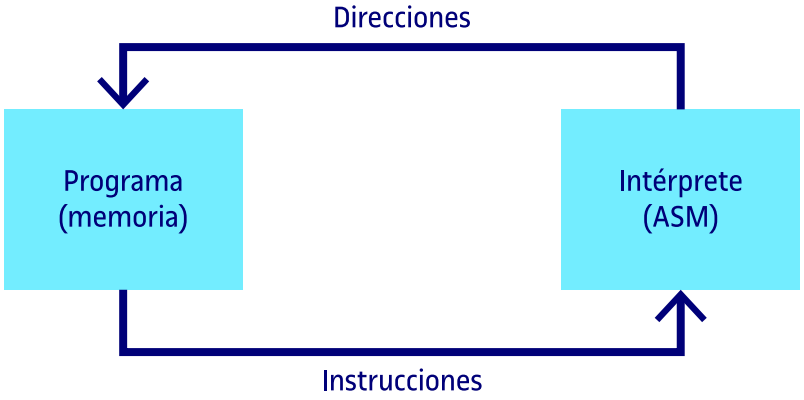
Función crítica

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 otras funciones dependen de ella o porque tiene mucha influencia en los parámetros que determinan el coste o la eficiencia finales del algoritmo global.

Así pues, es conveniente usar máquinas más genéricas, que puedan ejecutar conjuntos de funciones diferentes según la aplicación a que se destinen o la misma evolución del producto en que estén empotradas. Para hacerlo, los algoritmos tendrían que estar almacenados en módulos de memoria, de forma que las máquinas llevarían a cabo los algoritmos que, en un momento determinado, estuvieran guardados.

Una **máquina de estados algorítmica genérica** sería aquella capaz de interpretar las instrucciones del programa (los pasos del algoritmo) que hubiera en la memoria correspondiente.

Figura 26. Esquema de la relación entre la memoria de programa y la ASM de interpretación



La ASM que interpretara las instrucciones almacenadas en la memoria correspondiente seria, de hecho, la unidad encargada de procesar el programa en memoria. Para poder construir esta máquina es necesario determinar qué instrucciones tiene que interpretar y cómo se codifican en binario. A modo de ejemplo, en el subapartado «Ejemplo de máquina de estados algorítmica genérica» se describe una máquina de este tipo.

3.1.1. Ejemplo de máquina de estados algorítmica genérica

Las ASM que interpretan programas almacenados en memoria son genéricas porque son capaces de ejecutar cualquier programa, aunque está claro que no lo pueden hacer con la eficiencia de las máquinas específicas. Se convierten, pues, en máquinas de procesamiento de programas o **procesadores**. En este apartado veremos uno de muy pequeño: el Femtoproc.

La idea es detallar cómo se puede construir uno de estos intérpretes aprovechando lo que se ha visto en los apartados anteriores y, con ello, ilustrar el funcionamiento interno de los procesadores.

El Femtoproc será una pequeña ASM que interprete un repertorio mínimo de instrucciones: una para sumar (`ADD`), un par para hacer 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.

Nota

Como curiosidad, este repertorio permite hacer cualquier programa, por sofisticado que sea (sin embargo, el número de instrucciones 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 la NOT y la 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	$operando_1$			$operando_0$		

Instrucción	Bits							
	7	6	5	4	3	2	1	0
JZ	1	1	*dirección de salto*					

Los operandos de las operaciones ADD, AND y NOT se obtienen de un banco de registros que, teniendo en cuenta la codificación (se usan 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 que no se puede escribir nada (de hecho, son falsos registros). *R*1 tiene el valor 01h (unidad), de forma 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.

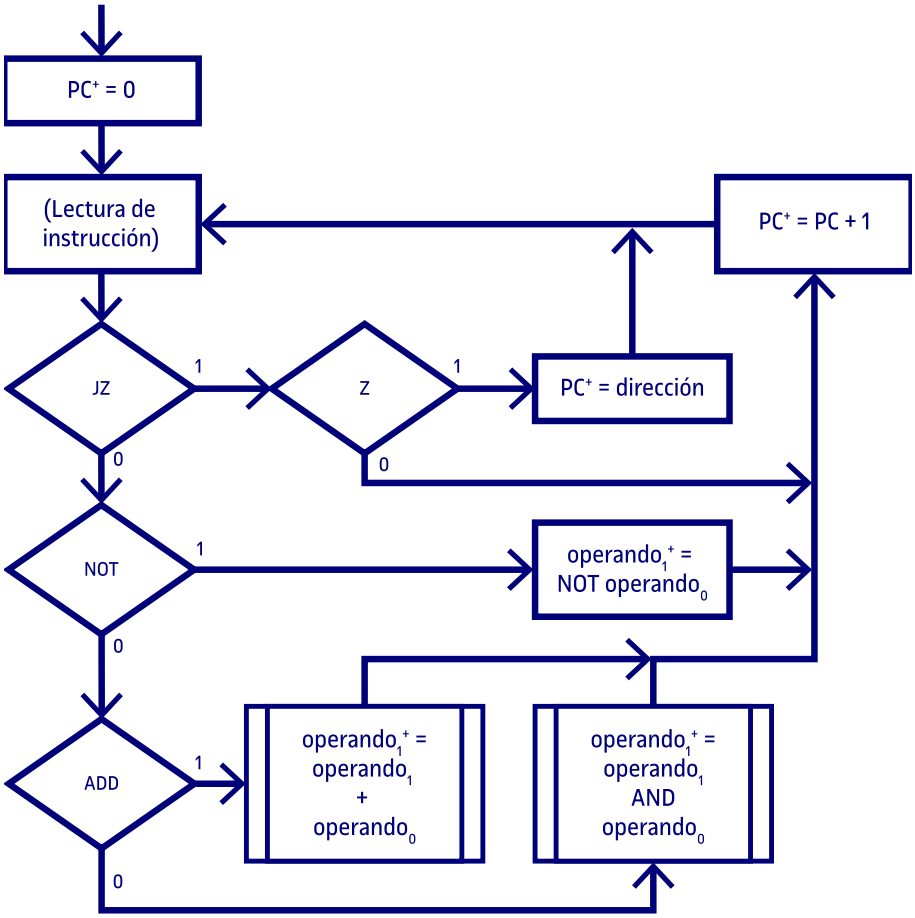
El resultado de las operaciones se almacenará en el $operando_1$. Si es *R*0 o *R*1, el resultado se perderá, pero el bit que identifica si la última operación ha sido cero o no (*Z*) sí que se actualizará. *Z* identifica un biestable que almacena esta condición.

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 hacer el algoritmo de interpretación de instrucciones y materializarlo con la ASM correspondiente. En el diagrama de la figura 27, los nodos de procesamiento de la suma y de la AND llevan una doble barra vertical a cada lado para indicar que pueden corresponder a más de un estado.

En este algoritmo se usa un registro que hace de contador de programa, o *program counter* (PC) en inglés. Se trata de un registro que contiene la posición de memoria de la instrucción a ejecutar. Como se ve en la ASM de la figura 27, 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 27. Diagrama de la ASM de interpretación de { ADD, AND, NOT, JZ }



El diagrama anterior se puede detallar un poco más, sabiendo 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:

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.
JZ?	$q_7 \cdot q_6$	Cálculo del bit que indica si la instrucción es *JZ*.
Z?	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}.
ADD	$q'_7 \cdot q_6$	Determinación de que la instrucción sea ADD.

Proceso o condición	Operaciones	Efecto
$operando_1^+ = operando_1 + operando_0$	0: $B^+ = BR[Q_{2-0}]$; 1: $BR[Q_{5-3}]^+ = BR[Q_{5-3}] + B$;	Cálculo en dos etapas, porque se accede dos veces a BR, de 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;	Cálculo en dos etapas de 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}.

En la tabla anterior, el nodo de procesamiento correspondiente a la lectura de la instrucción corresponde al estado en que se puede consultar la salida *M*[*PC*] de la memoria, puesto que el *PC* toma un valor nuevo justo después del estado anterior, sea cual sea (de puesta a 0 o de incremento). La asignación de *M*[*PC*] a *Q* se hace para simplificar las referencias a la señal de entrada *M*[*PC*], que pasa a ser *Q*.

Como se verá, en máquinas más complejas, la codificación de las instrucciones hace que sea necesario almacenar el código en una variable para ir descodificándolas de manera progresiva.

También hay dos nodos de procesamiento, uno para la suma y el otro para el producto lógico (`AND`), que se tienen que hacer en dos etapas, es decir, en una serie de dos estados, si se usa un banco de registros con una entrada y una única salida de datos, puesto que es necesario disponer de un registro auxiliar (*B*) que almacene temporalmente uno de los operandos.

La materialización de la ASM correspondiente se muestra a continuación solo a modo de ejemplo. Como se verá a continuación, el diseño de estas máquinas es más un arte que una ciencia, puesto que hay que transformar los diagramas correspondientes hasta llegar a los de partida para la implementación.

Implementación del Femtoproc

Si se partiera del diagrama anterior, se obtendría una máquina que funcionaría correctamente pero sería ineficiente. En concreto, si se asignara un estado por nodo de procesamiento, la máquina tendría nueve estados: «PC^+=0», «PC^+=dirección», «PC^+=PC+1», «lectura», «NOT», 2 para «ADD» y 2 para «AND».

Simplificando estados. El estado inicial «PC^+=0» solo tiene que poner el registro *PC* a 0, lo que ya se hace con el *reset* inicial. Por lo tanto, se puede suprimir. Ya son ocho.

Mirando las operaciones de los nodos de procesamiento del ADD y del AND se puede observar 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 siete.

En la figura 28 se puede comprobar esta simplificación: se ha reducido el número de nodos y se han minimizado los esquemas de cálculo. En la figura se usa *Rd* (de «registro de destino») para representar el registro en posición Q_{5-3} del banco de registros (BR), que sustituye al símbolo $operando_1$ y *Rf* (de «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 es $q_7 \cdot q'_6$, lo que se puede transformar en averiguar primero si $q_7 = 1$ y, entonces, según q_6 se sabe 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 tiene que almacenar en el registro *B*. El segundo paso sí que es diferente.

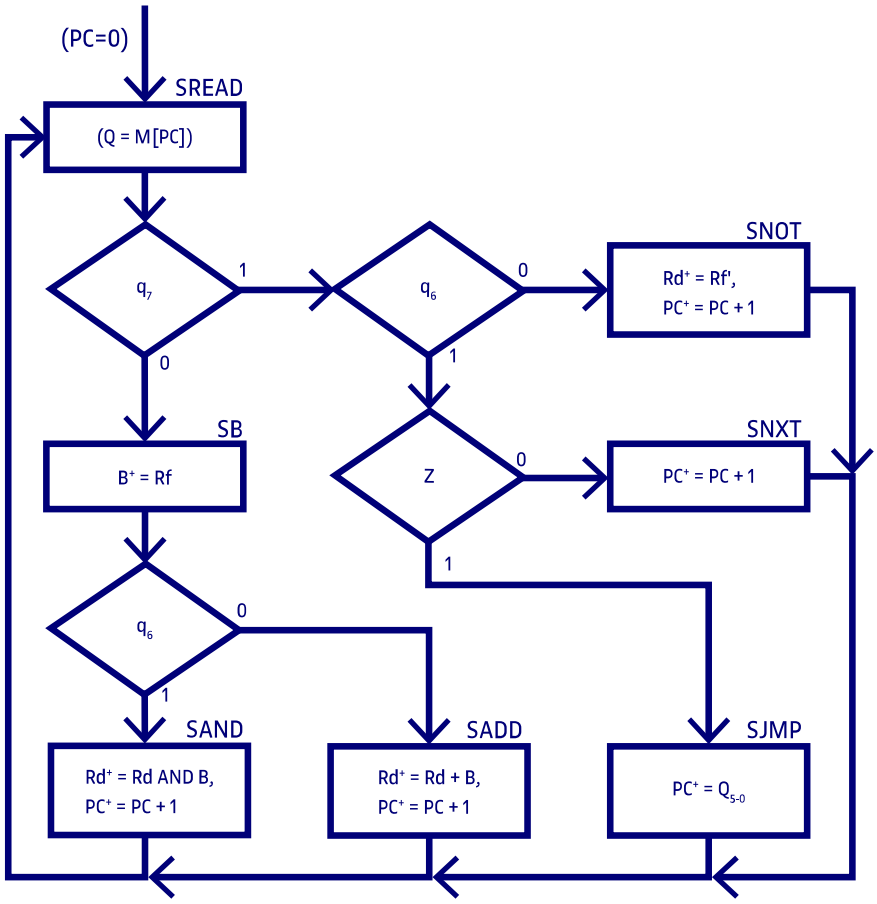
El camino de datos necesita los recursos que se enumeran a continuación:

- 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 cuales, los dos primeros con valores fijados.

- De cálculo: dos sumadores (uno para incrementar el *PC* y el 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 a ejecutar, $Q = M[PC]$).

- De conexión: buses, un multiplexor de selección del próximo valor para el *PC*, que puede ser $PC + 1$ o $Q_{5\text{-}0}$, otro del resultado a cargar en el BR y otro para elegir qué registro se quiere leer, si *Rd* o *Rf*.

Figura 28. ASM del Femtoproc adaptado para la materialización



La codificación de los estados será de tipo *one-hot bit*, de forma que la unidad de control se pueda implementar directamente a partir del diagrama, tal 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 denominan con «selOp» previo al nombre del recurso de memoria al que proporcionan la entrada de datos. En el caso del banco de registros, *selOpBR* hace referencia a la selección del resultado a 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 de la dirección de salto, que es $Q_{5\text{-}0}$.

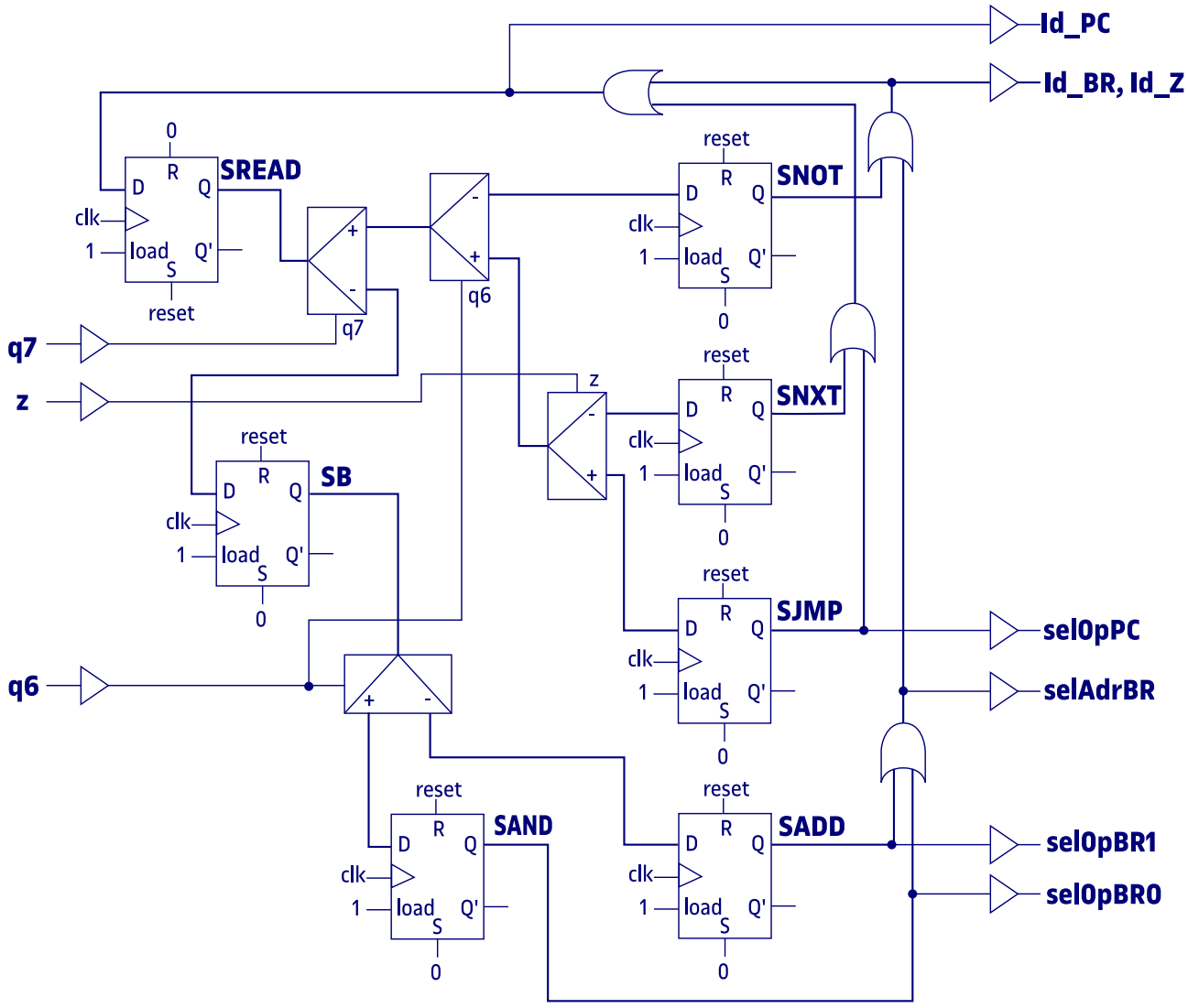
Estado	ld_PC	selOpPC	ld_B	selOpBR	ld_BR	selAdrBR	ld_Z
SREAD	0	0	x	xx	0	x	0

Estado	*ld_PC*	*selOpPC*	*ld_B*	*selOpBR*	*ld_BR*	*selAdrBR*	*ld_Z*
SNXT	1	0	x	xx	0	x	0
SJMP	1	1	x	xx	0	x	0
SB	0	x	1	xx	0	0	x
SNOT	1	0	x	00	1	0	1
SAND	1	0	x	01	1	1	1
SADD	1	0	x	10	1	1	1

Para la implementación de la unidad de control, la mayoría de salidas se pueden obtener directamente de un único bit del código de estado, salvo las señales *ld_BR*, *selAdrBR* y *ld_Z*, que se tienen que construir con sumas lógicas de estos bits.

En la figura 29 hay el circuito de control del Femtoproc. Es importante tener presente que no hay señal de arranque (*start*): este circuito funciona de manera autónoma ejecutando un bucle infinito (es como se denomina, a pesar de que siempre se puede reiniciar con *reset* o cortar la alimentación de corriente).

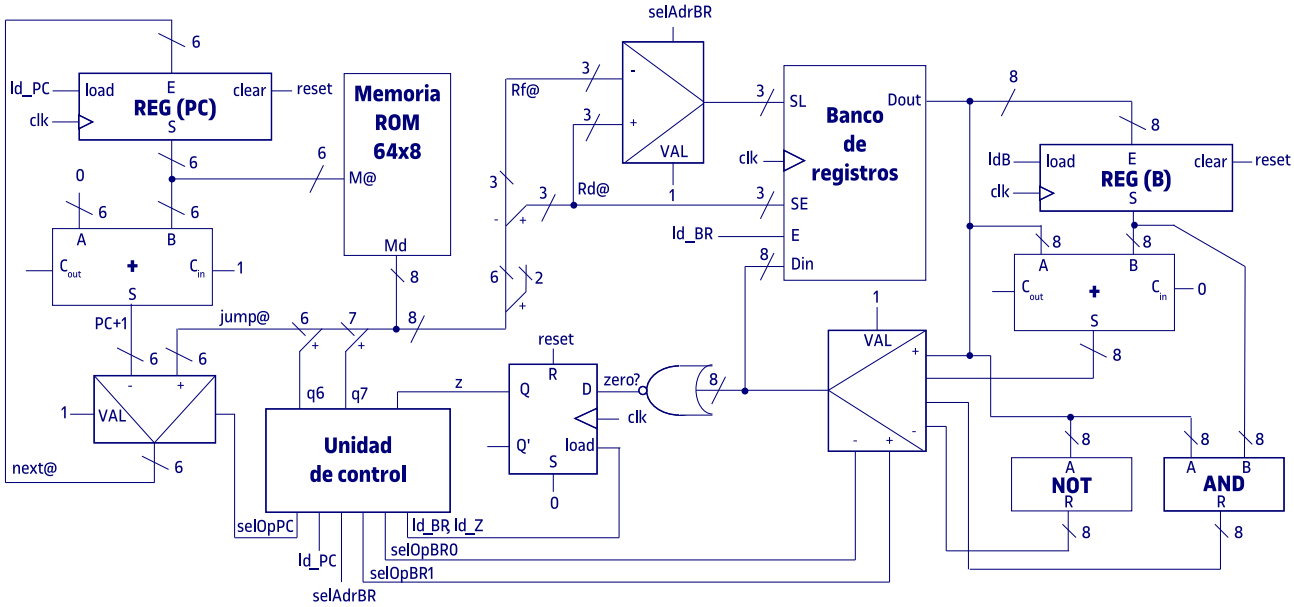
Figura 29. Unidad de control del Femtoproc, con codificación de estados de tipo *one-hot bit*



La unidad de control aprovecha las sumas lógicas que implementan los arcos de vuelta al nodo del estado SREAD para generar las señales *selAdrBR* y *ld_BR* (o *ld_Z*).

Finalmente, pues, el esquema del circuito completo del Femtoproc se puede ver en la figura 30.

Figura 30. Esquema completo del Femtoproc, con la unidad de control como módulo



Si se observa el dibujo en el sentido de lectura del texto que contiene, a la izquierda de la unidad de control hay el circuito para actualizar el *PC*, que depende de *selOpPC*. A la derecha hay 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 debajo de *BR* es el que elige qué operación hay que hacer en la instrucción en curso y, por lo tanto, (*selOpBR1*, *selOpBR0*) decide qué valor se tiene que almacenar en BR[*Rd*] y que se tiene que 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*.

Como se ha comentado, con esta máquina de interpretación se podría ejecutar cualquier programa construido con las instrucciones que puede descodificar. En el ejemplo siguiente se muestra un programa para calcular el máximo común divisor de dos números.

En este caso, se supone que los registros *R*6 y *R*7 se han sustituido por los valores de entrada del programa (es decir, son registros solo de lectura) y que el resultado queda en *R*5. Hay que tener presente que, inicialmente, todos los registros, excepto *R*0, *R*1, *R*6 y *R*7, están en 0.

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

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

La tabla siguiente muestra el contenido de la ROM del Femtoproc para el cálculo del MCD. Las primeras entradas, en que en la columna «Instrucción» pone «configuración», son explicativas y no de contenido. Básicamente, establecen qué registros se usarán de entrada de datos desde el exterior y cuáles de salida. En el resto de filas hay la dirección y el contenido de la memoria. La dirección se expresa con números en hexadecimal, de ahí que se les ponga una *h* al final. En el supuesto 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 a hacer (ADD, AND, NOT o JZ) y los que representan los operandos,

que pueden ser registros o direcciones. El nombre de los registros empieza por «R» y lleva un número que identifica la posición en el banco de registros BR.

Dirección	Instrucción	Codificación	Comentario
---	configuración	–	*R*0 = 100001 000010 = 80h, bit de signo
---	configuración	–	*R*1 = 000010 000011 = 01h, bit unitario
---	configuración	–	*R*5, registro de salida.
---	configuración	–	*R*6, registro de entrada.
---	configuración	–	*R*7, registro de entrada.
00h	ADD R2, R7	00 010 111	Pasa uno de los valores a un registro de trabajo.
01h	JZ 12h (end)	11 010010	Si es 0, el MCD es el otro número.
02h	ADD R3, R6	00 011 110	Pasa el otro valor a un segundo registro auxiliar.
03h	JZ 12h (end)	11 010010	Si es 0, el MCD es el primer número.
sub,04h	NOT R4, R2	10 100 010	$R4^+ = \text{NOT}(R2)$
05h	ADD R4, R1	00 100 001	$R4^+ = -R2 = \text{Ca2}(R2) = \text{NOT}(R2) + 1 = R4 + R1$
06h	ADD R4, R3	00 100 011	$R4^+ = R3 - R2 = R3 + R4$
07h	AND R0, R4	01 00001 100	Comprueba el bit de signo, el resultado no se guarda, porque *R*0 es solo de lectura.
08h	JZ 0Dh (pos)	11 001101	Si es positivo (si $R3 > R2$) 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 *R*2: $R2^+ = -(R3 - R2)$
0Bh	AND R0, R1	01 00001 001	Fuerza que el resultado sea cero.
0Ch	JZ 04h (sub)	11 00001100	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.
end,12h	ADD R5, R2		
13h	ADD R5, R3		$R5^+ = R2 + R3$, pero uno de los dos es cero.
14h	AND R0, R1		
hlt,15h	JZ 15h (hlt)		En espera, hasta que se le haga *reset*.

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 más grande.

Actividades

Actividad 10

Completad la columna de la codificación en la tabla anterior.

Actividad 11

Localizad, en el programa anterior, la secuencia de instrucciones que permite copiar el valor de un registro a otro.

Actividad 12

¿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 que se codifican los programas que interpreta una determinada máquina. El lenguaje máquina de la ASM 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 por ejemplo C++ o C, previa traducción. De todos modos, 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 incorporar los que se usan 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 tendría que convertir en un programa en lenguaje máquina que evaluara la condición de repetición y pasara a la ejecución del bloque. Así pues, una cosa como por ejemplo:

  ```
  mientras (c < f) hacer B;
  ```

  se tendría que 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 para poder llevar a cabo operaciones comunes de los programas de alto nivel de manera eficiente. Por ejemplo, sería muy costoso traducir una suma de una serie de números, puesto que serían necesarias tantas instrucciones como números

a 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 tiene que ser más completo, lo que implica que el algoritmo de interpretación sea más complejo y que la misma máquina sea más compleja, puesto que debe haber más recursos para poder llevar a cabo todas las operaciones del repertorio. Así pues, habría que tener, como mínimo, una instrucción de lenguaje máquina para cada una de las operaciones aritméticas, lógicas y de movimiento más habituales. Además, tiene que incluir una variedad más elevada de saltos, incluido uno de incondicional. De este modo, 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.

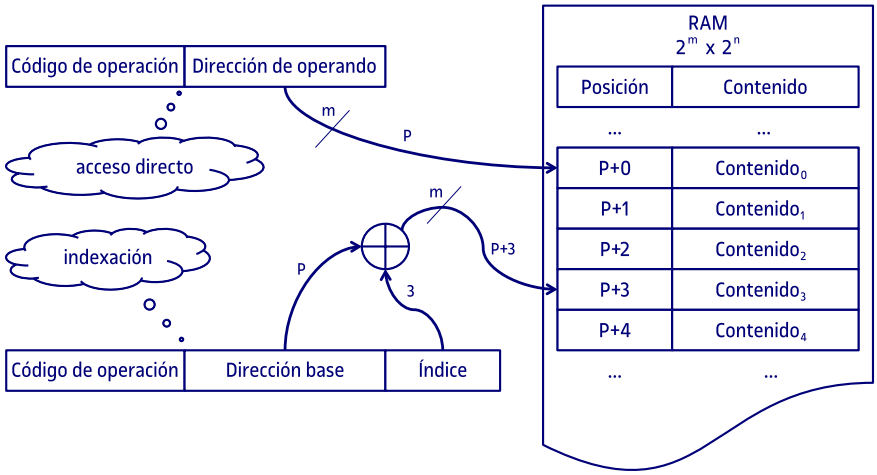
La memoria de programas debe tener mucha más capacidad y, consiguientemente, las direcciones deben ser de más bits. En este sentido, hay que tener en cuenta que la codificación de las instrucciones también se hace con más bits, puesto que el repertorio se ha ampliado.

El espacio reservado a los datos también tiene que crecer. Por un lado, 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 usar un acceso directo con su dirección. Conviene que haya, además, mecanismos de acceso indexado en 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 este modo, la manipulación de los datos por parte de los programas en lenguaje máquina es más eficiente. En la figura 31 hay un ejemplo de cada uno, con un código de instrucción que incluye el código de la operación que hay que hacer, la dirección del operando (P) y, para el acceso indexado, un índice que vale, en el ejemplo, 3.

Figura 31. Ilustración de los accesos directo e indexado al operando de una instrucción



Tomando como referencia el caso del Femtoproc, la memoria de programación es de tipo ROM. En cambio, las memorias para datos tienen que permitir tanto lecturas como escrituras. En este sentido, si se tuviera que mejorar esa máquina, se le tendrían que incorporar memorias para instrucciones y para datos diferenciados.

Este modo de construir las máquinas, con una memoria para las instrucciones y otra para los datos, se denomina **arquitectura Harvard**.

La arquitectura Harvard

La arquitectura Harvard se denomina así porque refleja el modelo de construcción de una de las primeras máquinas computadoras que se hizo: el IBM Automatic Sequence Controlled Calculator (ASCC), denominada Mark I, en la universidad de Harvard alrededor de 1944. Dicha máquina tenía físicamente separado el almacenamiento de las instrucciones (en una cinta perforada) del almacenamiento de los datos (en un tipo de contadores electromecánicos).

Sin embargo, a veces puede resultar más sencillo tener tanto las instrucciones como los datos en la misma memoria. De este modo, la máquina solo tiene que gestionar los accesos a una única memoria y, adicionalmente, puede ejecutar tanto programas pequeños que usen muchos datos como otros de muy grandes que traten pocos.

Las máquinas construidas de forma que usen una misma memoria para datos e instrucciones siguen la **arquitectura de Von Neumann**.

La arquitectura de Von Neumann

La arquitectura de Von Neumann se describe en *First Draft of a Report on the EDVAC*, con fecha 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 llamada Electronic Discrete Variable Automatic Computer (o computador electrónico automático de variables discretas) que se basa, sobre todo, en el trabajo hecho alrededor de otro de los primeros computadores: 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 suficientemente grande para incorporar tanto las instrucciones como los datos de los programas que tienen que ejecutar.

3.2.1. Máquinas de estados algorítmicas de procesadores

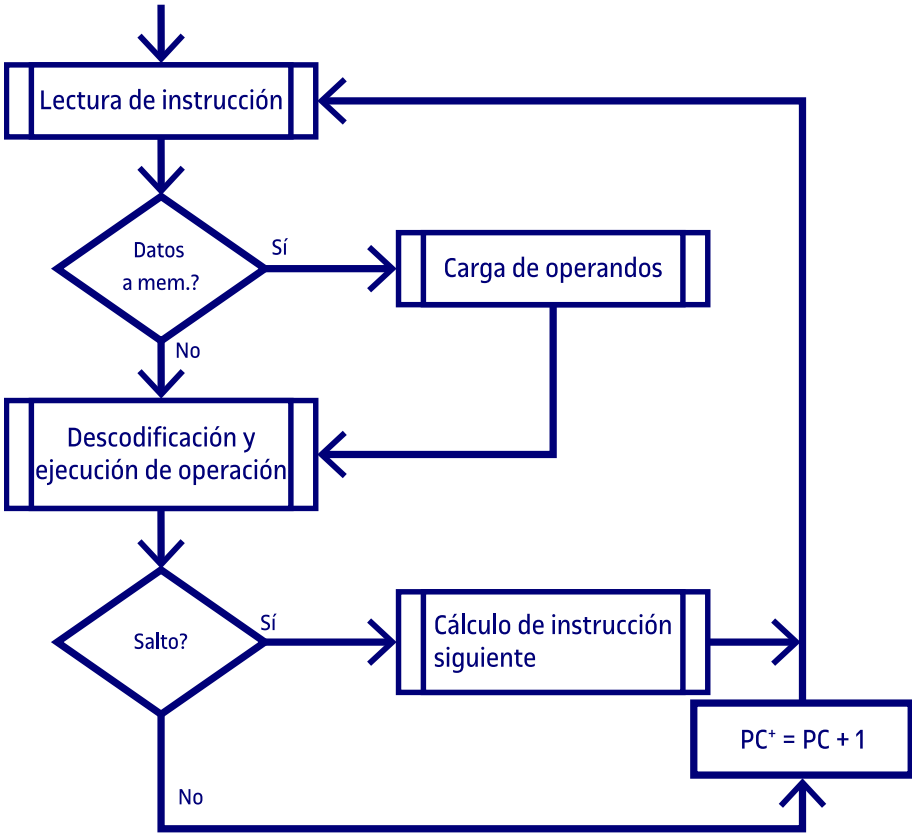
Una máquina de este estilo todavía se puede ver como una ASM de interpretación, pero el algoritmo que interpreta las instrucciones del lenguaje máquina es mucho más complejo que el que se ha visto anteriormente. En líneas generales, el intérprete trabaja de manera parecida, pero las fases del ciclo de 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 varias pasos, dado que el ancho en bits tiene que permitir codificar un repertorio grande, lo cual hace que pueda ocupar más de una palabra de memoria.

Además, en el caso de las instrucciones que trabajan con datos a memoria, se tiene que hacer la lectura, que puede no ser directa, y, consiguientemente, alargarse una serie de pasos que se incluyen en la fase de carga de operandos. Dado que también es conveniente que el abanico de operaciones que puedan llevar a cabo los computadores sea suficientemente grande, la descodificación de las mismas para poderlas ejecutar y su misma ejecución suelen requerir una serie de pasos que dependen del tipo de operación y, como consecuencia, hay 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 32 resume el flujo de control de una unidad de procesamiento que implementa un de estos repertorios de instrucciones. Cada etapa o fase se resuelve con una serie de pasos que se alargan durante unos cuantos ciclos de reloj. Por lo tanto, son nodos de procesamiento que incluyen ASM enteras dentro y, por eso, se marcan con una doble raya vertical a cada lado.

Figura 32. Diagrama del ciclo de ejecución de instrucciones



La implementación de estas ASM de interpretación se podría hacer de manera parecida a la que se ha visto, a partir de los diagramas correspondientes, pero teniendo en cuenta que la secuenciación de los pasos a seguir es muy compleja. Consiguientemente, 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 ASM correspondiente al caso del Femtoproc, la ejecución de las instrucciones dura entre uno y dos ciclos de reloj. Consiguientemente, las secuencias de pasos son bastante simples. En máquinas que interpreten instrucciones más complejas, las secuencias se alargan. Si se junta con el hecho de que la cantidad de combinaciones de entradas posibles para la parte controladora crece exponencialmente, puesto que los repertorios de instrucciones son más grandes (más bits para codificar las operaciones correspondientes) y, además, se usan muchos más bits de condición de la parte operativa (indicación de cero, acarreo o *carry*, desbordamiento, etc.), el número potencial de estados crece del mismo modo.

Por este motivo, es conveniente modelar la parte de control de estas máquinas con otras ASM encargadas de interpretarlas. En este caso, también serán ASM de interpretación de instrucciones, pero de un repertorio más limitado. Por ejemplo, una instrucción para cada tipo de nodo: una para los de procesamiento y otra para los de decisión.

Para distinguir esta máquina de la 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 hacen 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 microinstrucciones se denomina, obviamente, *microprograma*.

Los procesadores que interpretan repertorios de instrucciones complejos suelen estar microprogramados, es decir, la unidad de control correspondiente es una ASM implementada como intérprete de microprogramas.

3.2.2. Máquinas de estados algorítmicas microprogramadas

Como se ha comentado, el repertorio de microinstrucciones suele ser muy reducido. Por ejemplo:

- 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 hecha con datos provenientes de diversos registros fuente.

- Las de decisión sirven para hacer saltos condicionales. Por ejemplo, una microinstrucción de salto condicional activaría las microórdenes para seleccionar qué condición se tiene que cumplir y la de carga del contador de programa.

También hay 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 tienen que ejecutar en aquel ciclo.

En el cuadro siguiente hay 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 lo suficientemente grande para poder incorporar todas las posibles microórdenes del camino de datos y las direcciones de la memoria de microprogramación, que debe tener suficiente capacidad para almacenar el microprograma de interpretación del repertorio de instrucciones.

Micro-instrucción	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 que está organizado de una manera similar a los anteriores, pero con un único recurso de cálculo programable, para el que hay que proporcionar la operación que se hace en una microinstrucción concreta.

Esta arquitectura es muy común en la mayoría de 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 ASM que interprete un repertorio de microinstrucciones como el anterior permitiría ejecutar un microprograma de control de otra ASM que sería, finalmente, el procesador del repertorio de instrucciones de un lenguaje máquina concreto. Dado que se trata de una ASM muy sencilla que se ocupa de ejecutar las microinstrucciones en una secuencia determinada, se suele denominar *secuenciador*.

A pesar de que 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 usa 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 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.

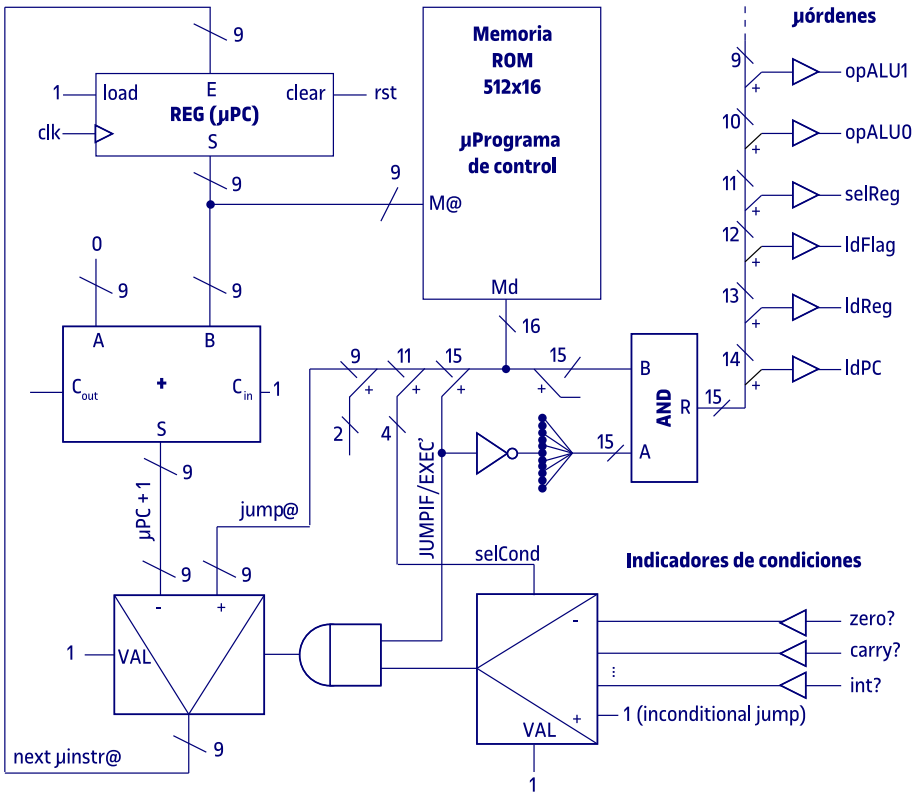
En la figura 33 hay el esquema de un circuito secuenciador para el repertorio de microinstrucciones que se acaba de comentar. Se trata de un secuenciador paralelo, en que cada microorden tiene un bit asociado. Por este motivo, este tipo de secuenciadores usan memorias con anchos de palabra muy grandes. En la práctica, suele compensar tener una codificación más compacta que reduzca el ancho a cambio de usar circuitos para descodificar las microinstrucciones.

La memoria ROM contiene una codificación binaria del microprograma que ejecuta una ASM de interpretación de instrucciones particular. Cada palabra de la memoria sigue el formato que se ha detallado en la tabla anterior.

En el supuesto de que se trate de una microinstrucción de tipo EXEC, los bits $Md_{14\text{-}0}$ son las microórdenes que están asociadas. Las señales correspondientes se han identificado con nombres que ejemplifican los tipos de microórdenes

que puede haber en una máquina tipo: *ldPC* para cargar un nuevo valor al 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 tiene que hacer la ALU; *ldFlag* para actualizar los biestables que almacenan condiciones sobre el último resultado (si ha sido cero, si se ha producido acarreo, entre otros), y así hasta 15. Al mismo tiempo que se suministran estas señales a la parte operacional, se calcula la dirección de la microinstrucción siguiente, que es la que se encuentra en la posición de memoria siguiente. Por eso, el contador de microprograma (*µPC*) se incrementa en uno: el multiplexor que genera *next_µinstr* selecciona la entrada *µPC*+1 cuando se está ejecutando una microinstrucción EXEC.

Figura 33. 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 forma que, cuando JMPIF/EXEC' (Md_{15}) sea 1, las microórdenes sean 0. En este caso, los bits $Md_{14\text{-}0}$ no son microórdenes sino que representan, por un lado, la selección de la condición (*selCond*, que es $Md_{14\text{-}11}$) que hay que tener en cuenta para efectuar o no el salto en la secuencia de microinstrucciones, y de la otra, la dirección de la microinstrucción siguiente (*jump@*, que es $Md_{8\text{-}0}$) en 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 hacer el salto. Estos bits de condición indican si el último resultado ha sido cero (*zero?*), si se ha producido acarreo (*carry?*) o si hay que interrumpir la ejecución del programa (*int?*), entre otros. La entrada de datos en 1 (constante) del multiplexor correspondiente sirve para hacer 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 haya la puerta AND previa a generar el control del multiplexor que genera *next_µinstr*. Con un secuenciador como este, sería posible construir controladores para ASM relativamente complejas: sus diagramas 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 ASM 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 almacenados en la memoria principal o provenir del exterior a través 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, de forma 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 manipulación de bits y, finalmente, las de control de flujo. Como es habitual, las operaciones aritméticas se hacen 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 tiene que hacer la ALU y los operandos que participan, entre otros. El argumento para acceder al segundo operando, si hace falta, se encuentra en un byte adicional. Así pues, hay instrucciones que ocupan un byte, y otras, dos.

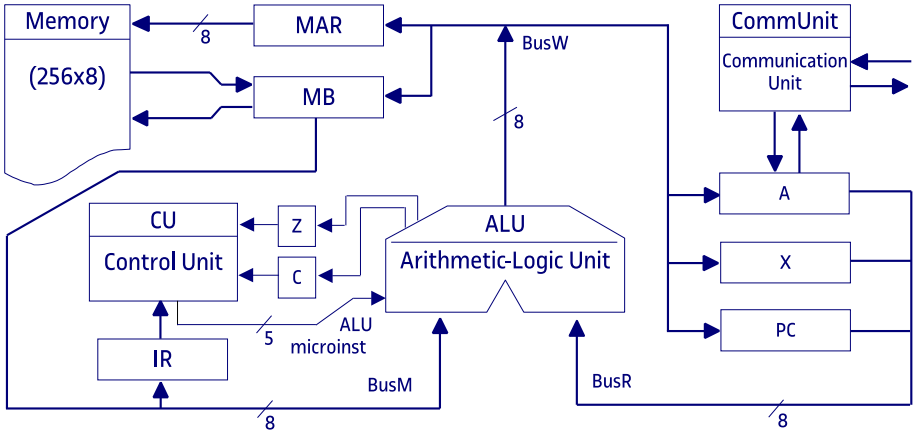
Formatos de las instrucciones de los lenguajes máquina

Los formatos de las instrucciones de los lenguajes máquina suelen ser muy variables porque ajustan los anchos de bits al tipo de operación que hacen y a los operandos que necesitan. Así, es habitual tener codificaciones en las que hayan instrucciones que ocupen un único byte con otros que necesiten dos, tres, cuatro o más. En el lenguaje máquina del YASP, las hay que ocupan un byte, si no necesitan ningún dato adicional, y las hay

que ocupan dos. En el primer caso hay, por ejemplo, las instrucciones que trabajan con registros. En el segundo, las que trabajan con datos en memoria.

En el esquema de YASP que hay a continuación se pueden ver todos los elementos que forman parte de este y cómo 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 hace a través del 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 los del código de la instrucción en curso, que se almacena en *IR*, y los indicadores de cero (*Z*) y de acarreo (*C*).

Figura 34. Diagrama de bloques del procesador elemental YASP



Para poder procesar los datos, dispone de un registro de 8 bits llamado *acumulador* (o *registro A*), puesto que es el registro en 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 a través del 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, hace falta guardar su 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 antes, también se puede dar la dirección de una posición de memoria donde habrá guardada la dirección de este operando. Es lo que se conoce como *indirección*, por el hecho de no dar directamente la dirección del operando.

Sin embargo, a veces resulta conveniente trabajar con una serie de datos de los cuales 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 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*.

Resumiendo, las instrucciones del lenguaje máquina de YASP tienen cuatro formas diferentes de obtener la dirección del segundo operando de una operación diádica (con dos operandos, uno de los cuales es el registro *A*):

- **Inmediata.** El valor del operando se especifica junto con la operación.
- **Directa.** La dirección del operando se indica con la operación. Hay que buscar el valor en la posición indicada.
- **Indirecta.** La posición de memoria donde hay el operando está guardada en la dirección que acompaña la operación. Es decir, se da la dirección donde hay la dirección real del operando.
- **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 denominan *direcciones efectivas* para distinguirlas de las que se dan en las instrucciones del programa que se ejecuta; y las formas de obtener las direcciones efectivas se denominan *modos de direccionamiento*. En concreto, se han descrito los modos inmediato, directo, indirecto e indexado.

Habiendo obtenido el segundo operando, la ALU puede hacer 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 monàdica se guarda a continuación.

En el caso de esta máquina elemental, la ALU hace 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 hacer 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. Sin embargo, a veces 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, puesto que los otros 3 son necesarios para codificar la operación. Consiguientemente, no se puede saltar a cualquier posición de memoria, sino que solo se puede ir a una instrucción situada 16 posiciones adelante o 15 atrás. Esto es así porque se suma un valor en Ca2 en el rango [–16, 15] al *PC* incrementado. Estas instrucciones utilizan un **modo de direccionamiento relativo**; es decir, la dirección que se adjunta se suma al contenido del *PC*. Por lo tanto, la dirección de la instrucción donde se salta se da en relación con la dirección de la instrucción siguiente a la del salto condicional.

Sea como fuere, 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 forma que la unidad de control pueda decidir qué tiene que hacer y en qué orden.

A pesar de que YASP es un procesador sencillo, es capaz de interpretar un repertorio bastante completo de instrucciones gracias a una unidad de control bastante compleja, que se puede materializar de forma directa si se microprograma.

Usando 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» hay el símbolo que identifica una posición concreta de la memoria de microprogramas.

La columna que contiene las microinstrucciones detalla qué transferencias de registros se harán en cada ciclo de reloj, si se trata de EXEC. Las cargas simultáneas de valores en *MB* y en *PC* son posibles porque usan 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 tipos JMPIF, el primer argumento es la condición que se comprueba, y el segundo, la dirección de salto, en caso de que sea cierta.

En este ejemplo hay 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.

Etiqueta	Código de operación	μ-instrucción	Comentario
	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	MAR^+ = MB + X	Direccionamiento indexado.
	JMPIF	*Inconditional*, DO	
SKIP:	EXEC	MAR^+ = MB	
	EXEC	MB^+ = M[MAR]	Direccionamiento directo.
	JMPIF	*Direct*?, DO	
	EXEC	MAR^+ = MB	
	EXEC	MB^+ = M[MAR]	Direccionamiento indirecto.
DO:	JMPIF	ADD?, X_ADD	**Fase 3.** Descodificación, 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

Actividad 13

Sin considerar las operaciones posibles de la ALU, que se codifican con 5 bits, tal como se muestra en la figura 34, indicad qué microórdenes harían falta 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 llamar *ld_PC*.

Actividad 14

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 una cosa como *BusW* = *BusR* + 1.

3.3. Procesadores

Tal como ya se ha comentado en el subapartado «Máquinas de estados algorítmicas genéricas», las ASM 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 diversa, puesto que depende del repertorio de instrucciones que tengan que interpretar y de la función de coste que se quiera minimizar.

En este subapartado 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** (a veces se abrevian como *μarch* o *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 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 mismo bloque de instrucciones. En este sentido, es razonable que las instrucciones trabajen con datos almacenados en un banco de registros y que algunos faciliten la implementación de varias formas 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.), porque usan 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 que son más fáciles de descodificar. Normalmente, son bastante reducidos en comparación con los CISC, lo cual hace que reciban el nombre de 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/a memoria, aunque tengan variantes según los operandos que admitan. Este tipo de arquitecturas se denomina *load/store*, y se suelen oponer a otros en 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, puesto que tienen problemas comunes: acceso a memoria por lectura de instrucciones, acceso a memoria por lectura/escritura de datos, procesamiento de los datos.

3.3.2. Microarquitecturas con *pipelines*

Para ir más rápido en el procesamiento de las instrucciones, se pueden tratar varias formando un *pipeline*: un tipo de cañería por donde pasa un flujo que es tratado en cada segmento de la misma, de forma que no hay que esperar que se procese totalmente una porción del flujo para tratar otra de diferente.

En las unidades de procesamiento, los *pipelines* suelen formarse con las diferentes fases del ciclo de ejecución de una instrucción. En la figura siguiente, hay un *pipeline* de una máquina similar a YASP, pero habiendo reducido los operandos a los inmediatos y directos. Así pues, el ciclo de ejecución de una instrucción sería:

- Fase 1. Lectura de instrucción (LI).
- Fase 2. Lectura de argumento (LA), que puede ser vacía si no hay argumento.
- Fase 3. Cálculo de operando (CO), que hace una nueva lectura si es directo.
- Fase 4. Ejecución de la operación (EO), que puede hacer varias cosas, según el tipo de instrucción: un salto, actualizar el acumulador o escribir a memoria, entre otras.

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 en la figura 35, con el *pipeline*, se podría tener una de acabada a cada ciclo de reloj, sacado de la latencia inicial de 4 ciclos.

Figura 35. *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
Periodo	1	2	3	4	5	6	7
	Latencia						

Los *pipelines* son, pues, una opción para aumentar el rendimiento de las unidades de procesamiento y, por este motivo, la mayoría de microarquitecturas los incluyen.

Ahora bien, también presentan un par de inconvenientes. El primero es que hacen falta 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 esté y, también, que hay que «vaciar» el *pipe* después 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, puesto que depende de los saltos que se hagan en un programa.

Actividad

Actividad 15

Rehaced la tabla de la figura 35 suponiendo que la segunda instrucción es un salto condicional que sí que se efectúa. Es decir, que después de la ejecución de la instrucción la siguiente es otra de diferente de la que se encuentra a continuación. Con vistas 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 hacer varias operaciones simultáneamente. Por ejemplo, se pueden incluir varias ALU que faciliten la ejecución paralela de las diversas etapas de un *pipeline* o de varias operaciones de una misma instrucción. En este último caso, el paralelis-

mo 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 otro lado, esta solución de múltiplos ALU permite de avanzar la ejecución de una serie de instrucciones que sean independientes de una 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 de ellas con su propio *pipeline*, si es el caso.

Esta microarquitectura *multi-core* es adecuada para procesadores que ejecutan varios programas en paralelo, puesto 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, lo que se hace es construirlos de forma que la parte procesadora tenga recursos de memoria suficientes para trabajar de manera eficiente (es decir, que el número de instrucciones ejecutadas por unidad de tiempo sea tan grande como sea posible), pero que no incluya el grueso de la información (datos e instrucciones), puesto que se puede almacenar mejor con tecnología específica.

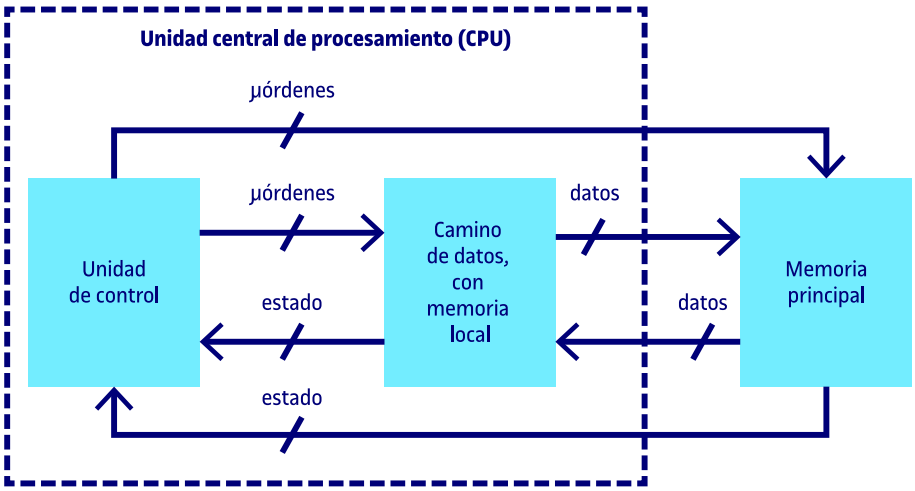
Así pues, el modelo de construcción de los procesadores debe tener en cuenta los acondicionamientos tecnológicos que hay con vistas a su materialización. Generalmente, las memorias se construyen con tecnologías que incrementen su capacidad. Los circuitos de procesamiento, en cambio, se hacen con unos criterios muy diferentes: poder procesar muchos datos con tan poco tiempo como sea posible y, en consecuencia, la tecnología es diferente. Esto hace que las microarquitecturas de los procesadores deban tener en cuenta que la unidad de procesamiento tendrá que 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 procesar la información. Se incluye también la unidad de control correspondiente.

- La **memoria principal** es la parte del procesador que se ocupa de almacenar datos y programa. Como se ha visto, puede estar organizada de forma que se vea como un bloque único (arquitectura de Von Neumann) o como

dos bloques que guardan datos e instrucciones por separado (arquitectura de Harvard).

Figura 36. 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 para contener la mayoría de los datos que se usan en un momento determinado, sin tener que acceder a memoria, con independencia de si se implementa un RISC o un CISC. En todo caso, por más grande que sea este banco de registros, no puede contener los datos necesarios para la ejecución de un programa cualquiera y se tiene que recurrir a las memorias externas a la CPU.

Las arquitecturas *load/store* se pueden construir sobre microarquitecturas optimizadas para hacer operaciones simultáneas de acceso a memoria y operaciones internas a la CPU. Normalmente, son construcciones en 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 llevar a cabo ejecuciones en paralelo, puesto que las mismas instrucciones imponen una secuencia en la ejecución de los accesos a memoria y la realización de las operaciones.

Con todo, hay microarquitecturas que separan la parte de trabajo con datos de la parte de trabajo con direcciones para poder materializar CPU más rápidas.

De manera parecida, también es habitual, para determinados procesadores, separar la memoria de datos de la de instrucciones. Así 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).

De todos modos, 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. Esto pasa, 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 usar con más probabilidad) en memorias más rápidas, aunque no tengan tanta capacidad. Estas memorias son las denominadas *memorias caché*, porque son transparentes a la unidad de procesamiento y a la memoria principal.

Memorias caché

Las memorias caché se denominan así porque son memorias que se usan como las madrigueras, para almacenar cosas a escondidas de la vista de los otros. En la relación entre las CPU y las memorias principales se ponen memorias caché, más rápidas que las principales, para almacenar datos e instrucciones de la memoria principal para que la CPU, que trabaja a más velocidad, tenga un acceso más rápido a ellos. Si no hubiese memorias caché, la transferencia de información se haría igualmente, pero a la velocidad de la memoria principal.

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 tanto pueden servir para gestionar una base de datos en una aplicación profesional como para jugar.

Por este motivo, suelen ser procesadores con un repertorio de instrucciones bastante amplio para ejecutar eficientemente tanto un programa que necesita hacer muchos cálculos como otro que tiene un requerimiento más elevado de movimiento de datos. A pesar de que 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 complejos de gestionar cuando hay saltos, por ejemplo.

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 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 concreta, 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 instrucciones específicas para el tipo de aplicación a que se destinará el procesador. Por ejemplo, puede tener de 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 tendrá que usar varias unidades aritmeticológicas (u otros elementos de procesamiento) en paralelo.

De hecho, el ejemplo anterior se corresponde con los procesadores de señales digitales o DSP (del inglés, *digital signal processor*). Los DSP, pues, están orientados a aplicaciones que requieran el procesamiento de un flujo continuo de datos que se pueda ver como una señal digital que hay que ir procesando para obtener una de salida convenientemente elaborada.

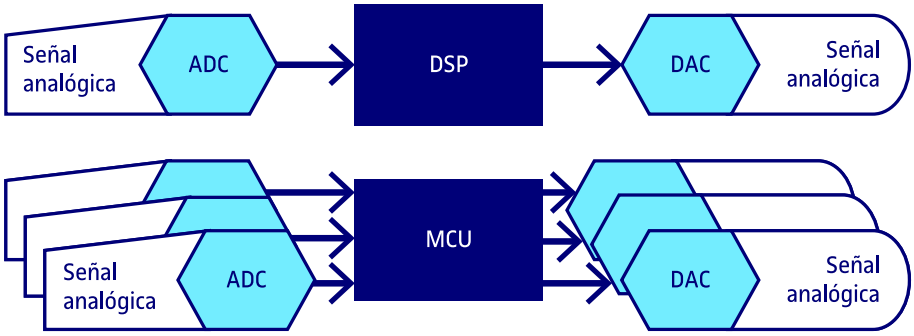
Para el caso particular de las imágenes, hay DSP específicos, denominados *unidades de procesamiento de imágenes* o GPU (del inglés *graphics processing units*) que tienen una ISA similar, pero que suelen usar un sistema de memoria diferente, con las direcciones organizadas en 2D para facilitar las operaciones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

Un caso de características muy diferentes es el de los controladores, que tienen que coger 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 muchos bits y su procesamiento individual no es especialmente crítico. Ahora bien, estas aplicaciones necesitan procesadores con un repertorio de instrucciones que incluyan una variedad grande de operaciones de entrada/salida y una microarquitectura que las permita realizar de manera eficiente.

En estos casos, los procesadores actúan como microcontroladores o MCU (del inglés *microcontroller unit*). Por tener la capacidad de captar datos del exterior y de sacar datos 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 que los primeros procesan pocas señales de muchos bits muy rápidamente y que los segundos son capaces de tratar muchas señales de pocos bits casi simultáneamente.

Figura 37. Ejemplificación de la dualidad entre DSP y MCU (ADC y DAC son, respectivamente, convertidores analógico-digital y digital-analógico).



Sin embargo, 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 ser denominados de una manera u otra según las características dominantes, pero no porque las otras no las tengan.

Además de implementar un repertorio específico de instrucciones, hay 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 mucha 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 algo más lentas.

Para ir rápido, los procesadores incluyen mecanismos de *pipelining* y de procesamiento paralelo. Además, suelen estar construidos según la arquitectura Harvard, para trabajar concurrentemente con instrucciones y datos. Para consumir menos, incluyen mecanismos de gestión de la memoria interna de la CPU que evite lecturas y escrituras innecesarias en la memoria principal. En general, 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 de forma autónoma. La interacción de los computadores no se limita a los usuarios, sino que también abarca el entorno o el sistema en que se encuentran. Así pues, hay computadores que se perciben como tales y que tienen un

modo de trabajo interactivo con los humanos: jugamos, creamos, consultamos y gestionamos información de todo tipo, trabajamos y hacemos casi cualquier cosa que nos pase por la imaginación. Y también los hay de más «escondidos», que controlan aparatos de todo tipo o que permiten que haya dispositivos de todo tipos que tengan un «comportamiento inteligente».

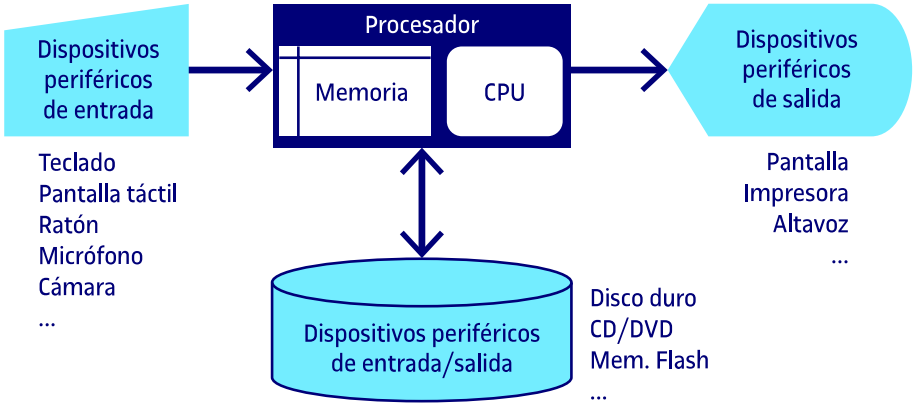
Sean como sean, los computadores tienen unos rasgos comunes que se comentarán en el resto del apartado.

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 para procesar (y también los programas para hacerlo) y hay que dotarlos 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 38. 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 debéis haber notado que hay bastantes más periféricos: altavoces, escáners (para capturar imágenes impresas), micrófonos, cámaras y una retahíla adicional que se amplía continuamente para ajustarse a las aplicaciones informáticas que puede llevar a cabo un computador.

También hay periféricos que pueden realizar las dos tareas. Es decir, pueden ocuparse de la entrada de los datos al 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,...) y de lectura de tarjetas de memoria. Otros periféricos de este tipo son los discos duros y las unidades de estado sólido (SSD, del inglés *solid state drive*), que están dentro de la caja del ordenador, y los módulos de conexión a red inalámbrica.

Generalmente, este tipo de periféricos se utilizan para almacenar datos que se pueden recuperar cuando un determinado proceso lo solicite y, igualmente, modificarlos si lo 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.

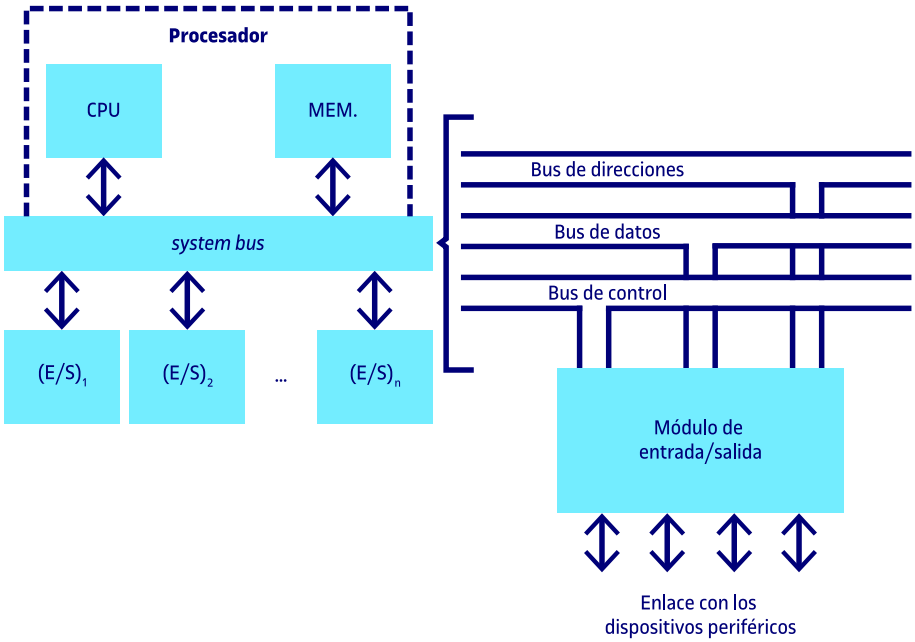
Para comunicarse con el procesador, los dispositivos periféricos disponen de controladores que, además de gobernar su funcionamiento, se relacionan con un **módulo de entrada/salida** del computador que sea. Estos módulos hacen de puente entre el periférico y el procesador, adaptando las diferentes maneras de funcionar, los formatos de los datos y la velocidad de trabajo. Por ejemplo, las lecturas de disco se hacen 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 velocidades de transferencia ajustadas a sus buses.

CommUnit del YASP

A modo de ilustración, hay que decir que la CommUnit del YASP hace las funciones de módulo de entrada/salida del procesador. Cualquier periférico que se conecte 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.

Una arquitectura básica de un computador, pues, tendría que tener varios módulos de entrada/salida conectados a un bus del sistema, que también tendría conectadas la CPU y la memoria principal, tal como se muestra en la figura 39.

Figura 39. Esquema de un computador basado en un único bus



El bus del sistema se organiza en tres buses diferentes:

- El bus de datos se usa para transportar datos entre todos los elementos que conecta. A menudo, los datos van de la memoria principal a la CPU, pero también a la inversa. Las transferencias entre CPU y módulos de E/S o entre módulos de E/S y memoria son relativamente más esporádicas.

- El bus de direcciones transmite las direcciones de los datos (y de las instrucciones) a que accede la CPU bien sea a la memoria o a algún módulo de E/S. También lo aprovechan los módulos de E/S para acceder directamente a datos en memoria.

- El bus de control transmite todas las señales de control para que todos los elementos se puedan comunicar correctamente. Desde la CPU, la unidad de control es la encargada de recibir las señales que le tocan y emitir las correspondientes a cada estado en que se encuentre. Es importante tener presente que, de hecho, cada módulo conectado al bus tiene su propia unidad de control.

El acceso a los periféricos desde el procesador se hace a través de 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, hay 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 que se pueden recoger datos para el periférico o almacenar temporalmente los que provienen de este.

Las operaciones de entrada/salida de datos programados implicarían sincronizar el procesador con el periférico correspondiente. Es decir, el procesador tendría que esperar la disponibilidad del periférico para hacer la transmisión de los datos, lo que querría decir una pérdida notable de rendimiento en la ejecución de la aplicación asociada. Como ejemplo ilustrativo, hay que tener en cuenta 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 hace que el acceso a los datos pueda tardar unos cuantos milisegundos, tiempo que sería perdido por el procesador.

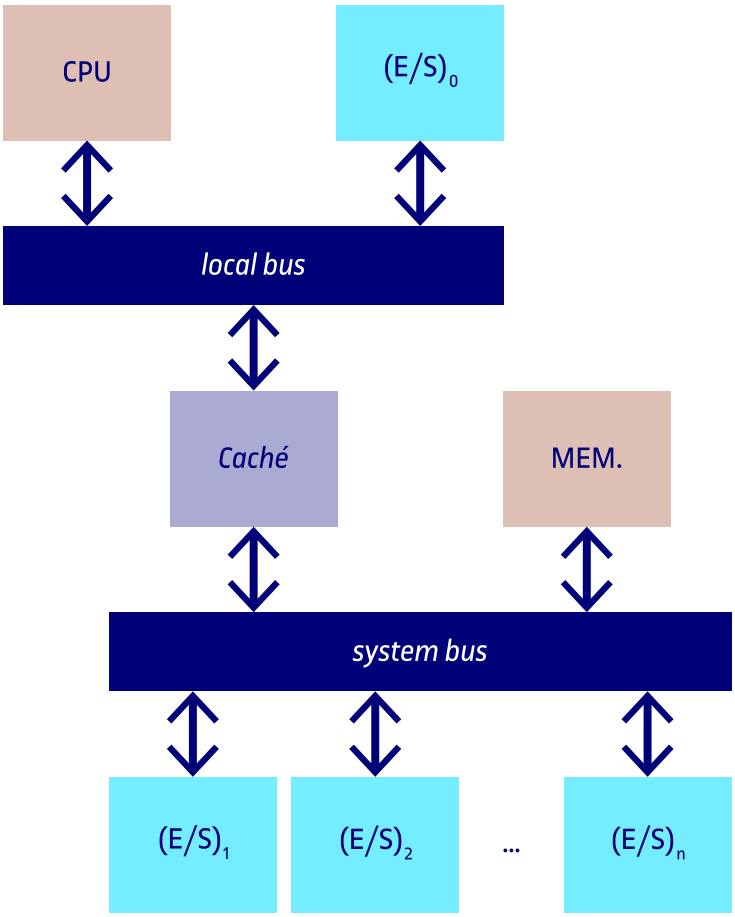
En general, la transferencia de datos entre periféricos y procesador se hace a través de módulos con mecanismos de acceso directo a la memoria (DMA, del inglés *direct memory access*). Los módulos dotados de DMA transfieren datos entre su memoria interna y la principal del computador sin intervención de la CPU. De este modo, 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 tiene acceso en memoria a través del bus del sistema, con prioridad para la CPU, obviamente.

Como la memoria principal todavía resulta relativamente lenta en cuanto a la CPU, es habitual interponerle memoria caché (figura 40). Tal 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 el esquema siguiente, 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 hacen a través de memoria. El bus local trabaja a más velocidad que el del 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 «Arquitecturas orientadas a aplicaciones específicas» se comentarán brevemente algunas ampliaciones posibles, según la aplicación a que se destinen los computadores correspondientes.

Figura 40. Esquema de un computador con memoria caché



3.4.2. Arquitecturas orientadas a aplicaciones específicas

Las arquitecturas genéricas se pueden extender 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 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 muchos 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 hacer 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 caché: las más rápidas y pequeñas cerca de la CPU y las más lentas y grandes, de la memoria principal.

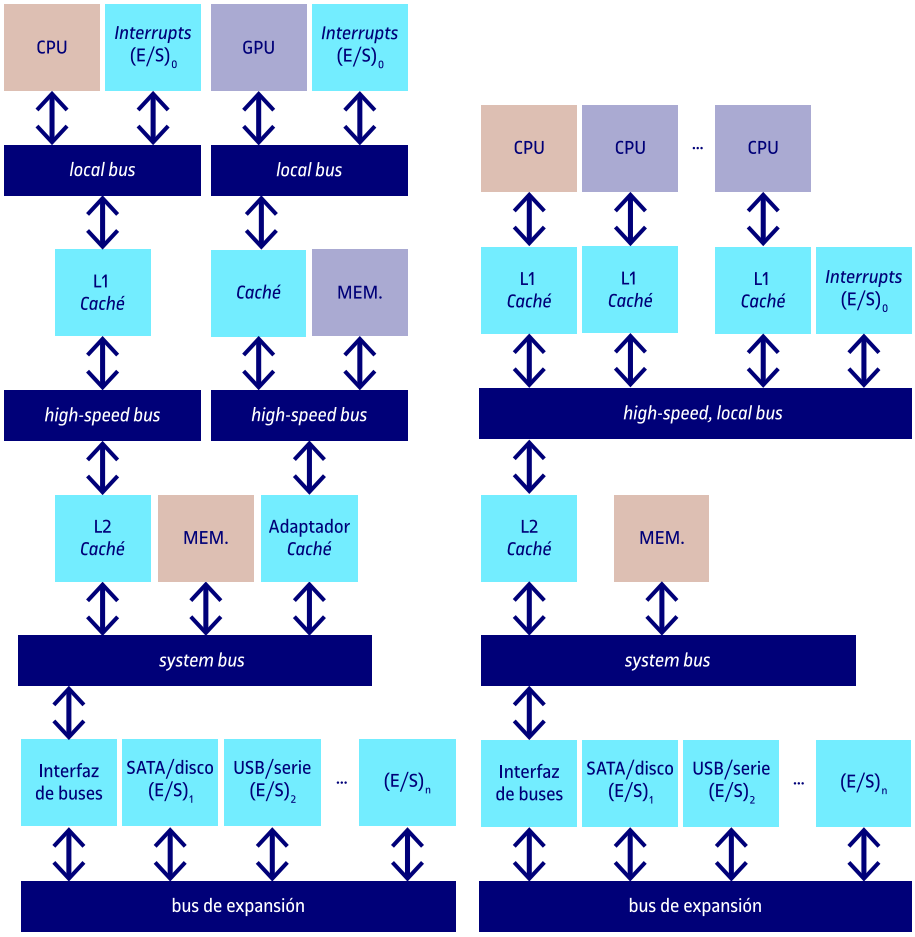
Las crecientes necesidades de procesamiento y memoria en todo tipo de aplicaciones han hecho que las arquitecturas originalmente concebidas para computadores de alto rendimiento sean cada vez más comunes en cualquier ordenador.

Computador de alto rendimiento

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

En la figura 41 se muestran dos arquitecturas específicas para aplicaciones gráficas y multiproceso. Las dos usan dos niveles de memoria caché, *caché* L1 y L2, y organizan el sistema en una jerarquía de buses. En la dedicada a aplicaciones gráficas, hay un coprocesador específico (una GPU) que tiene una memoria principal local y una memoria caché propias. La opción multiprocesadora se basa en diversas CPU que tienen una memoria caché local, pero que comparten una de general para la memoria principal.

Figura 41. 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 hace posible integrar cada vez más componentes en un único chip, lo que aporta la ventaja de poner más recursos para construir los computadores, pero también complica cada vez más la arquitectura. Por ejemplo, hay chips *multicore* 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.

Se pueden quitar los dispositivos periféricos de un computador sin que este pierda la esencia que lo define: la capacidad de procesar información automáticamente 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 denomina *arquitectura de Von Neumann*, en honor al modelo establecido por un computador descrito por este autor.

Se ha visto que los procesadores se pueden materializar 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 ASM también se pueden materializar exclusivamente como circuitos secuenciales y se pueden focalizar tanto en el procesamiento de la información que almacenan en sus variables como en el control de sistemas con muchas señales de entrada y salida.

Se ha visto que todas estas máquinas 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 cual facilita tanto su análisis como su síntesis.

Se ha tratado un caso paradigmático de las EFSM, el contador. El contador usa 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, pues, 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, también permitían expresar 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 demasiado complejos para los objetivos de este material: los que constituyen las máquinas que denominamos *computadores*.

Ejercicios de autoevaluación

Los problemas que se proponen a continuación se tendrían que resolver habiendo acabado el estudio del módulo, de forma que sirvan para comprobar el grado de logro de los objetivos indicados al principio.

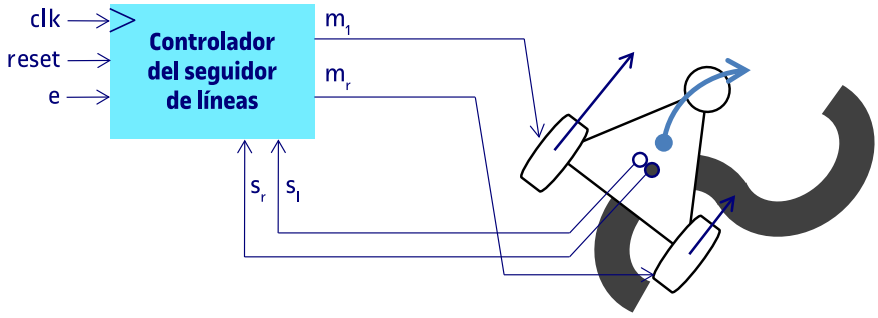
1. Este ejercicio debe servir para reforzar la visión de las FSM como representaciones del comportamiento de controladores. Para ello se pide que diseñéis el controlador de un robot seguidor de líneas.

El vehículo del robot (figura 42) se mueve por par diferencial (si las dos ruedas giran a la misma velocidad en el mismo sentido, el vehículo se mueve en linea recta en el sentido de giro y, si una va más lenta que la otra, gira hacia el lado en que se encuentre). El robot está dotado de un par de sensores en la parte de abajo que sirven para detectar si está sobre la línea de forma total o parcial. Las entradas del controlador son, pues, 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 m_l y m_r son las que controlan los motores izquierdo y derecho para hacer girar las ruedas correspondientes:

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 hacia delante

Los giros se tienen que hacer cuando, al seguir una línea, uno de los dos sensores no la detecta. Por ejemplo, si el sensor izquierdo deja de detectarla, se tiene que girar hacia la derecha.

Figura 42. Esquema de bloques del robot seguidor de líneas



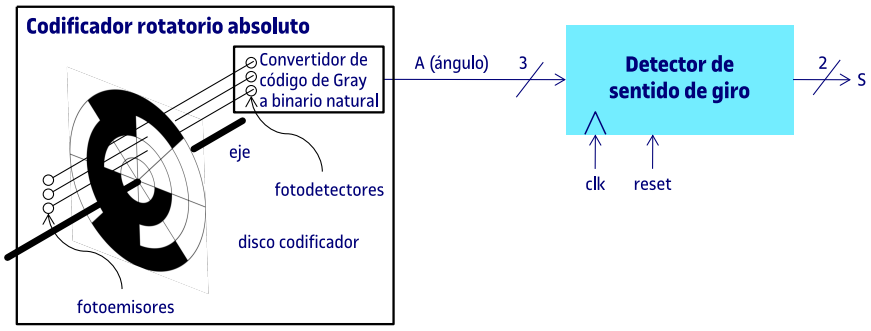
Se supone que, inicialmente, el robot estará localizado sobre una línea y que, en el supuesto de que no la detecte por ninguno de los sensores, se parará. Con esta información, se tiene que hacer el diagrama de la FSM correspondiente (no hay que hacer el diseño del circuito que lo implementará).

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. Tenéis que construir la EFSM de un detector de sentido de rotación de un eje e implementar el circuito correspondiente.

Este detector usa 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 a construir tendrá, como entrada, un número natural de 3 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 el sentido de las agujas 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.

Figura 43. Detector de sentido de giro con un codificador rotatorio absoluto



El esquema de la figura 43 presenta el sistema que hay que diseñar. El detector usa 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

Hay que tener presente que del sector [315º, 360º) se pasa directamente al sector [0 o, 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 como se ha presentado en la tabla anterior.

Los códigos de Gray se usan en este tipo de codificadores rotatorios porque entre dos códigos consecutivos solo cambia un bit, lo cual hace que el cambio de sector se detecte con este cambio. Si cambiaran más bits entre dos códigos consecutivos y el cambio no fuera simultáneo, entonces se codificarían sectores erróneos entre el inicial y el final.

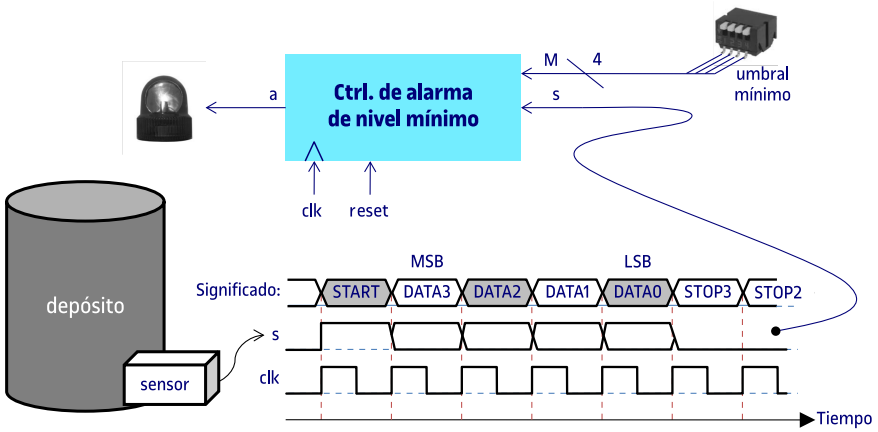
Para resolver este problema, hay que tener presente que el detector de sentido de giro debe comparar el sector actual con el anterior. Se supone que la cadencia de las lecturas de los ángulos *A* es lo bastante elevada para que no pueda haber saltos de dos sectores entre dos lecturas consecutivas.

3. En muchos casos, las EFSM incorporan varios contadores, por eso mismo es importando el ejemplo del contador del apartado 1.3. En este caso, tenéis que hacer el modelo de un controlador de una alarma de nivel mínimo de un depósito, que necesitará una variable de contador.

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 en forma de número binario natural de 4 bits. Este número indica el porcentaje de llenado del depósito, desde el 0 % ($000010_{(2)}$) hasta el 100 % ($1111_{(2)}$) y, por lo tanto, cada unidad equivale al 6,25 %. A fin de reducir cables, los datos del sensor se envían en series de bits. Tal como se muestra en la figura 44, primero se envía un bit a 1 y entonces 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 acabando por el menos significativo. Siempre habrá, como mínimo, 4 bits a cero siguiendo el último bit del número en una transmisión.

Figura 44. Controlador de alarma de nivel mínimo



Diseñad la EFSM 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 envíe datos cada vez que detecte algún cambio significativo en el nivel.

4. Este problema trata del uso de ASM en los casos en que hay un número elevado de entradas, de forma que la representación del comportamiento sea más compacta y comprensible: hay que hacer el modelo de control de un horno de microondas.

El horno tiene una serie de sensores que le proporcionan información a través de las señales que se enumeran a continuación:

- *s* indica que se ha pulsado el botón de inicio/parada con un pulso en 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 es cero cuando la rueda del temporizador está en la posición 0.

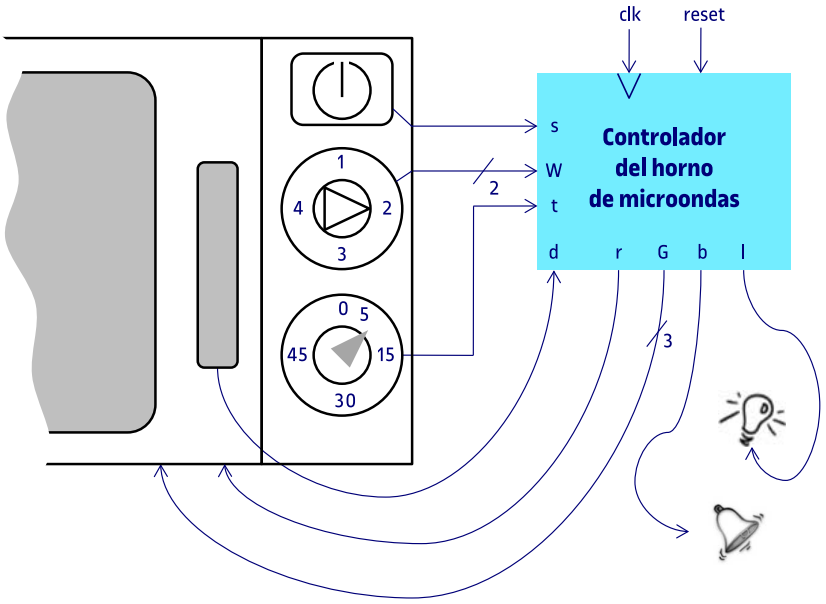
Hay que tener presente que el botón de inicio/parada se puede pulsar en cualquier momento y que la rueda del temporizador solo puede volver a la posición 0 cuando haya pasado el tiempo correspondiente o si el usuario fuerza su regreso girándola hasta aquella posición. Si se pulsa para poner en marcha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

El controlador toma la información proporcionada por estos sensores y hace las actuaciones correspondientes, según el caso, a través de las señales siguientes:

- *b* se debe poner a 1 durante un ciclo de reloj para hacer sonar un timbre de aviso de finalización 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* tiene que 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.

Con todo, el esquema del conjunto se puede ver en la figura 45.

Figura 45. Esquema de bloques de un horno de microondas



Se trata, pues, de que diseñéis la ASM correspondiente a un posible controlador del horno microondas que se ha descrito.

5. En muchos casos, hay que hacer cálculos complejos que hay que tratar independientemente con ASM específicas porque no se dispone de todos los recursos de cálculo necesarios para hacerlo en un único ciclo de reloj. En este ejercicio tenéis que diseñar una ASM para calcular:

$R = A \cdot X^2 + B \cdot X + C$

La máquina tiene una señal de entrada *start* que se pone en 1 cada vez que se quiere hacer un cálculo, una entrada X y una salida R para el resultado, pero solo dispone de un bloque multiplicador. También tiene tres variables con un contenido fijo, A, B y C y, además, podéis añadir las que os convengan.

Si X, A, B y C contuvieran números naturales de 4 bits, ¿cuáles serían los anchos de las variables adicionales que usáis y la de R?

6. En este ejercicio se trata de ver cómo se puede transformar una ASM en un circuito con arquitectura de FSMD. Así, tenéis que implementar 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:

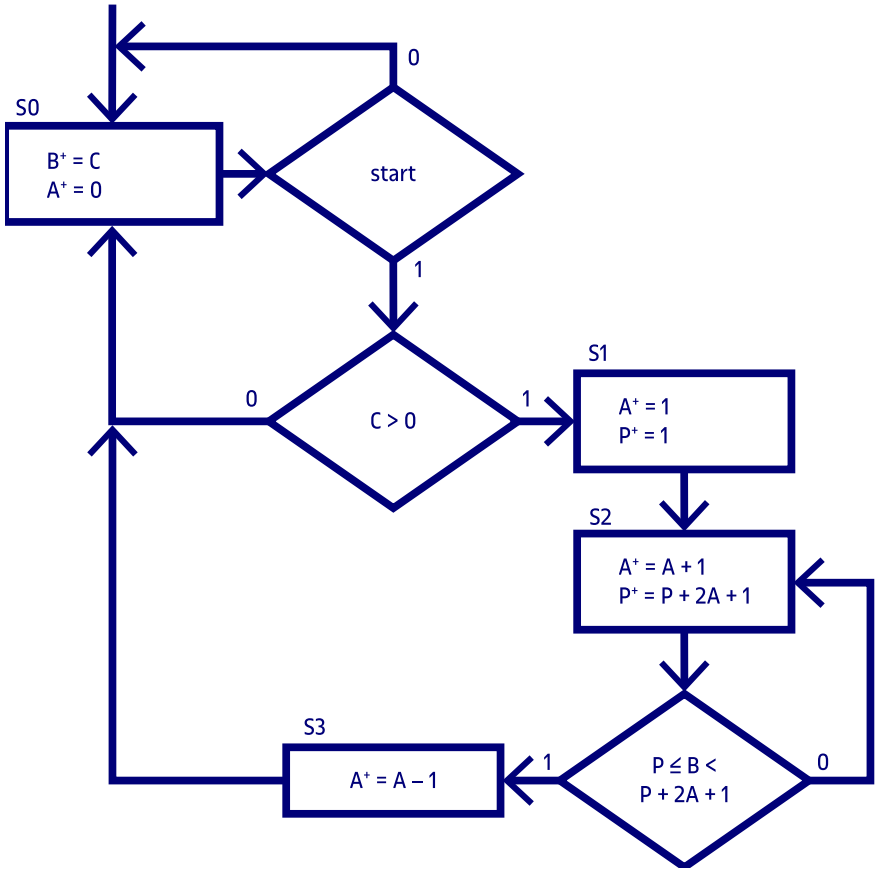
$A^2 \leq C < (A + 1)^2$

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 ASM de la figura 46 se puede ver el algoritmo correspondiente. 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 se calcula como $P + 2A + 1$.

Figura 46. ASM de cálculo de la raíz cuadrada entera



A partir del diagrama, explicad por qué es necesario el estado S_3.

Diseñad el circuito correspondiente, siguiendo la arquitectura de FSMD, con la unidad de control según el modelo visto que reproduce el diagrama.

7. Se trata ver cómo se pueden construir procesadores a partir de otros ampliando el repertorio de instrucciones. En este caso, tenéis que modificar el diagrama de la figura 28, que representa el comportamiento del Femtoproc, de forma que pueda trabajar con más datos que los que se pueden almacenar en sus 6 registros libres. Para ello, hay que tener presente que el nuevo repertorio tiene un formato de instrucciones de 9 bits, tal 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, respectivamente. En estos casos, el *operando*$_0$ identifica el registro (*Rf*) del banco de registros que se usará para guardar el dato o para obtenerlo. La dirección de la memoria

de datos que se usará 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 como se ha presentado en el módulo «Circuitos lógicos secuenciales», con una entrada para las direcciones (*M@*) de 4 bits y otra para indicar cuál es la operación que se debe hacer (*L'*/*E*) y con una entrada/salida de datos (*Md*). Se supone que la memoria hace la operación en un ciclo de reloj.

Hace falta, pues, que ampliéis el diagrama de la figura 28 para representar el funcionamiento de una ASM 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 con la instrucción que hay que ejecutar.

8. Indicad qué modificaciones harían falta para adaptar la microarquitectura del YASP (figura 34) a un repertorio de instrucciones capaz de trabajar con direcciones de 16 bits y, así, poder disponer de una memoria de hasta 64 K. 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. Esto incluye la de salto incondicional y excluye las instrucciones con modo de direccionamiento inmediato. ¿Qué se tendría que tocar en el camino de datos? ¿Qué implicaciones tendría en la unidad de control?

Solucionario

Actividades

1.

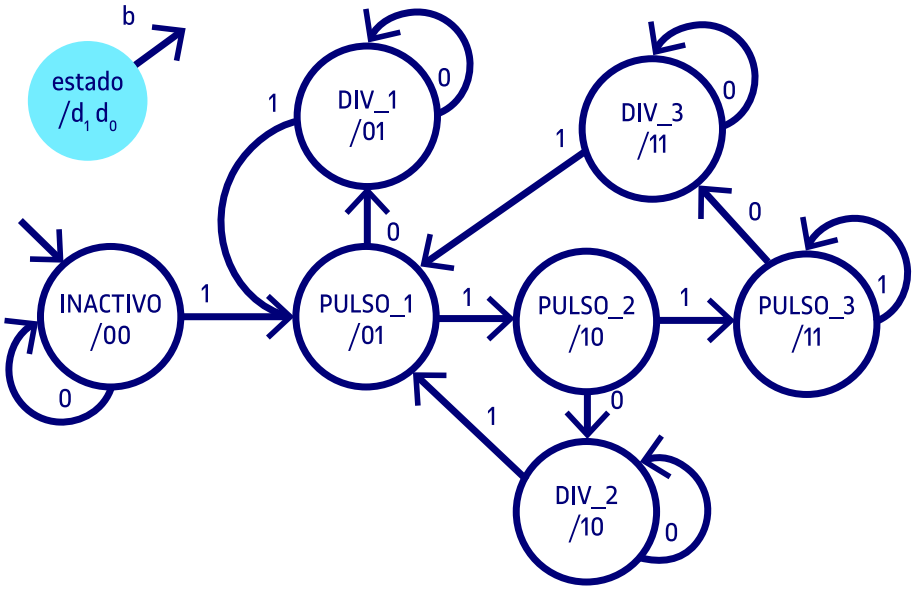
Para hacer el grafo de estados, hay que empezar por el estado inicial (INACTIVO) y decidir los estados siguientes según todas las posibilidades de combinaciones de entradas. En este caso, solo pueden ser o b=0 o b=1. Mientras no se detecte ningún cambio en la entrada (b=0), la máquina permanecerá inactiva. En el instante en que llegue un 1, hay que pasar a un nuevo estado para recordar que se ha iniciado un pulso (PULSO_1). A partir de ese momento, se harán transiciones hacia otros estados (PULSO_2 y PULSO_3) mientras b=1. De este modo, la máquina descubre si el ancho del pulso es de uno, dos, tres o más ciclos de reloj.

En el último de estos estados (PULSO_3) se tiene que mantener hasta que no acabe el pulso de entrada, tenga el ancho que tenga. En otras palabras, una vez b haya sido 1 durante tres ciclos de reloj, ya no se distinguirá si el ancho del pulso es de tres o más ciclos de reloj.

La máquina tiene que recordar el ancho del último pulso recibido, de forma que la salida D se mantenga a 01, 10 o 11, según el caso. Para lo cual, en los estados de detección de ancho de pulso (PULSO_1, PULSO_2 y PULSO_3), cuando se recibe un cero de fin de pulso (b=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 tiene que pasar a PULSO_1 cuando se recibe, de nuevo, otro 1 en la entrada.

Figura 47. Grafo de estados del detector de velocidad de transmisión



Del grafo de estados de la figura anterior se puede deducir la tabla de transiciones siguiente:

Estado actual				Entrada	Estado siguiente			
Identificación	q_2	q_1	q_0	b	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

Estado actual				Entrada	Estado siguiente			
Identificación	q_2	q_1	q_0	b	Identificación	q_2^+	q_1^+	q_0^+
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
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 hacer una codificación de los estados que respete el hecho de que el estado inicial sea el 00001, pero que la salida se pueda obtener directamente 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.

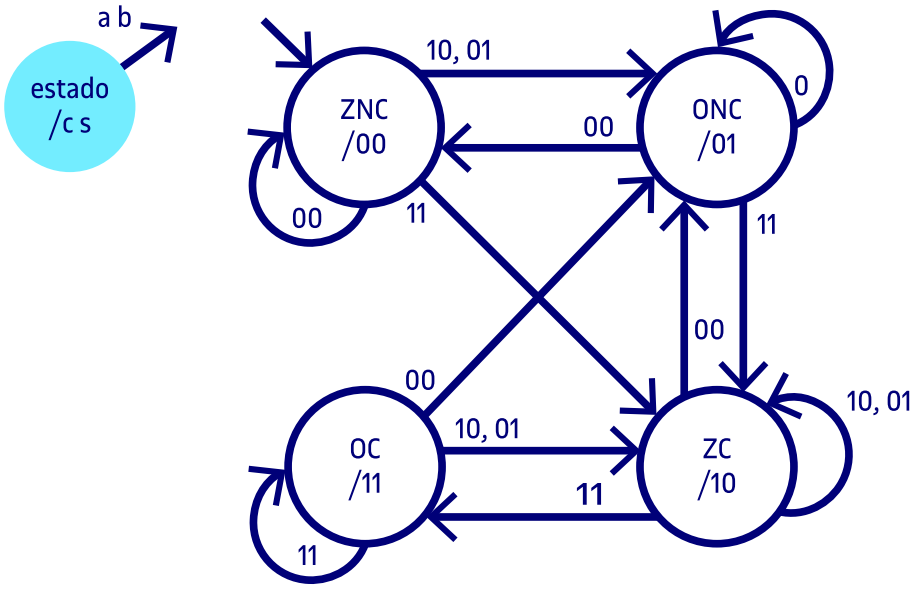
El estado de un sumador, en un momento determinado, lo fija tanto el resultado de la suma como del acarreo de salida del periodo de reloj anterior, que es el que hay que recordar. Por lo tanto, la máquina de estados correspondientes empieza en un estado en que se supone que la suma previa ha sido (c, s) = 0 + 0, donde '+' indica la suma aritmética de los dos bits.

En el estado inicial (ZNC, de *zero and no carry*) se queda mientras los bits a 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 lo 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. Por lo tanto, de ZNC pasa a ZC (*zero and carry*).

Una reflexión similar se puede hacer estando en ONC y en ZC. Hay que tener presente que, una vez hecha, se tiene que comprobar que se han previsto todos los casos de entrada.

En la figura 48 hay el grafo de estados correspondiente. Las entradas (a, b) se denotan de manera compacta con los dos bits consecutivos:

Figura 48. Grafo de estados del sumador en serie



Del grafo de estados de la figura anterior 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.

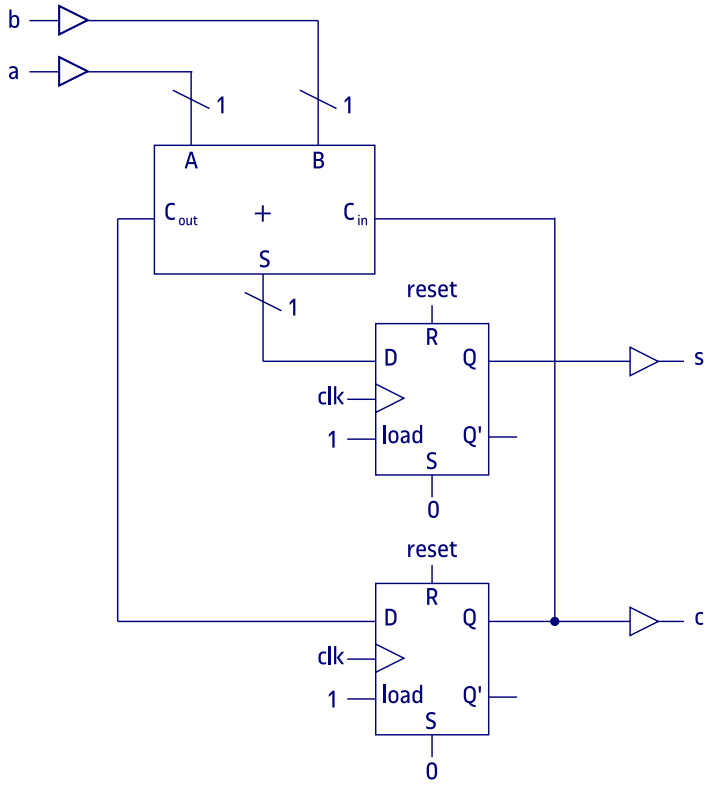
Las funciones de excitación son, de hecho, coincidentes con las de un sumador completo:

$q_1^+ = q_1\ (a \oplus b) + ab$

$q_0^+ = (a \oplus b) \oplus q_1$

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 49. Circuito secuencial de suma de números en serie



3.

Lo primero que hay que hacer son las tablas de transición y de salidas. En este caso, se opta para hacerlas junto con la codificación que se explica a continuación.

La codificación de los estados se hace 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 que representará la condición ON/OFF'. Y la condición $(T - R < 0)$ se expresará con una señal que se denominará *c*, que será 1 cuando se cumpla. Finalmente, hará falta un selector de valor para *P*, que será *sP*. Cuando *sP* sea 0, $P = 0$ y cuando sea 1, $P = f(T, R)$.

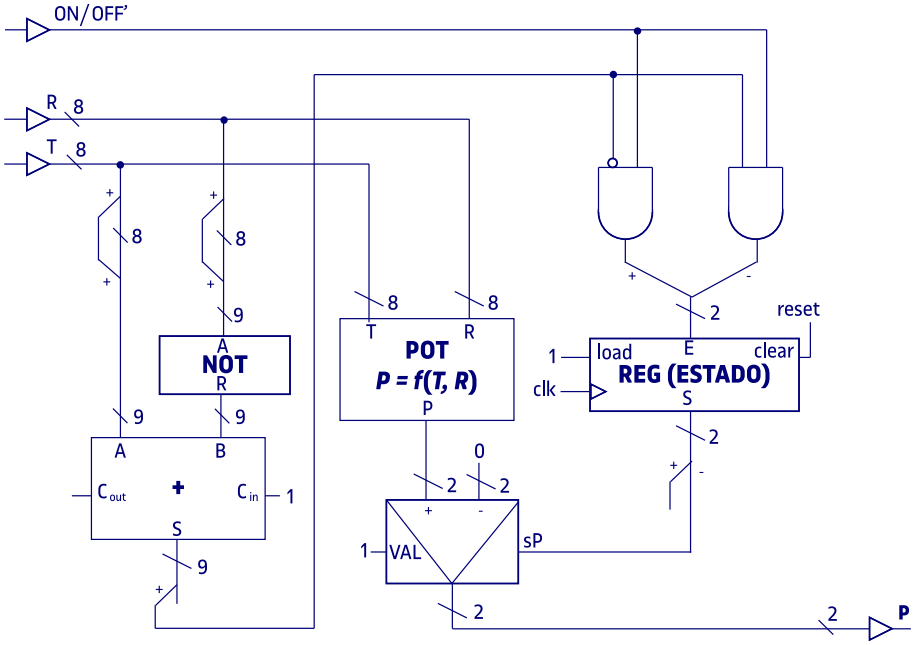
Del grafo de estados de la figura 9 se puede deducir la tabla de transiciones siguiente:

Estado actual			**Entradas**		**Estado siguiente**		
Identificación	q_1	q_0	**ON/OFF'**	***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	0	x	APAGADO	0	0

Estado actual			Entradas		Estado siguiente		
Identificación	q_1	q_0	**ON/OFF'**	c	**Identificación**	q_1^+	q_0^+
CALENTANDO	0	1	1	0	ENCENDIDO	1	0
CALENTANDO	0	1	1	1	CALENTANDO	0	1
ENCENDIDO	1	0	0	x	APAGADO	0	0
ENCENDIDO	1	0	1	0	ENCENDIDO	1	0
ENCENDIDO	1	0	1	1	CALENTANDO	0	1

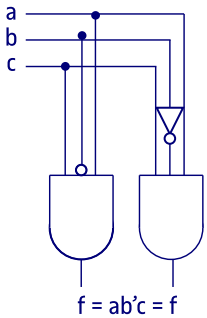
A partir de esta tabla se pueden obtener las funciones q_1^+ y q_0^+. La salida sP coincide con q_0, puesto que solo hay que poner una potencia del radiador diferente de cero en el estado de CALENTANDO, que es el único que tiene q_0 a 1. El esquema que hay a continuación, en la figura 50, es el del circuito secuencial correspondiente.

Figura 50. Controlador de un calefactor con termostato



Nota

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.

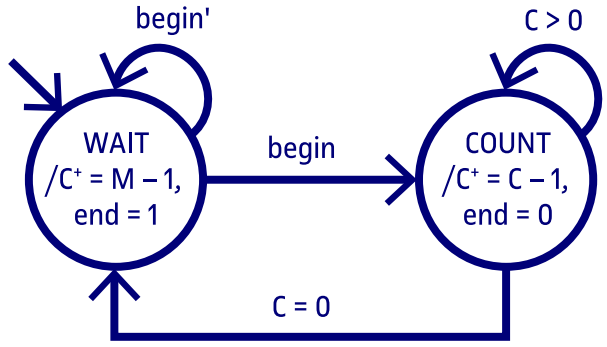


El cálculo de c se hace con el bit de signo del resultado de la operación $T - R$. Para garantizar que no haya problemas de rebosamiento (caso de una temperatura negativa, por ejemplo), se amplía el número de bits en 1, extendiendo el bit de signo.

4.

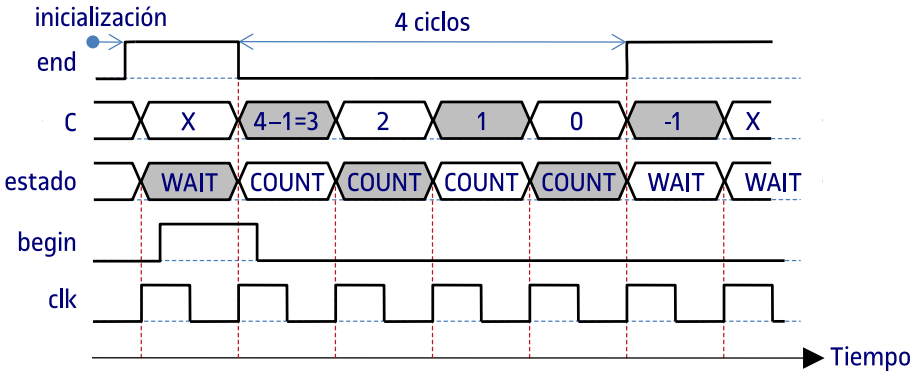
En este caso, en el estado de espera WAIT hay que ir efectuando cargas de valores decrementados en el registro de cuenta C hasta que alguna de estas cargas coincida con *begin* = 1. Entonces hay que pasar al estado de ir contando, COUNT. En este estado, se decrementa C y, si el último valor de C ha sido cero, se acaba la cuenta. El diagrama de transiciones de estados es, pues, muy similar al del contador de la figura 10.

Figura 51. Grafo de estados de un contador hacia atrás «programable»



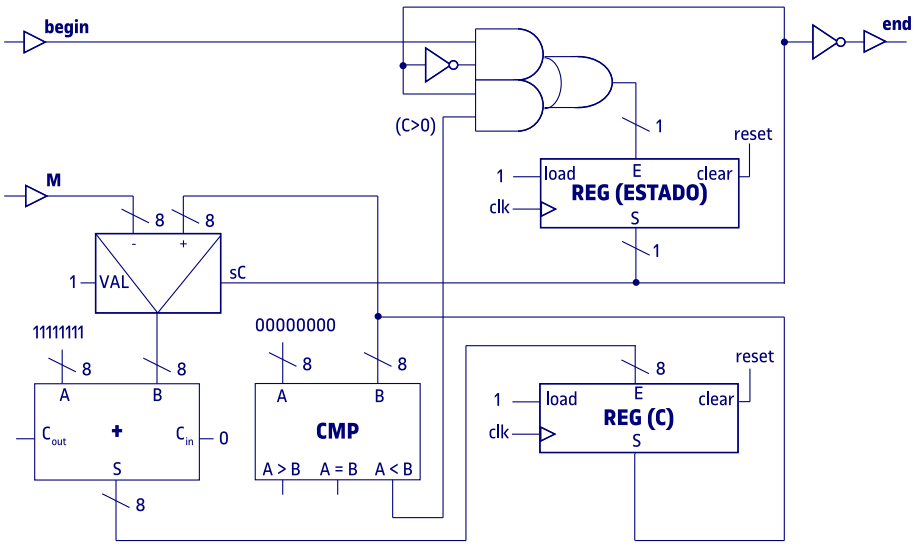
Hay que tener presente que, como pasa con el contador incremental, el registro C acaba cargando un valor más que la cuenta que hace. En este caso, pasa a tener el valor –1 justo al volver al estado WAIT. Este valor se pierde en el ciclo siguiente, con la carga de un posible nuevo valor a contar.

Figura 52. Cronograma de ejemplo para el contador haica atrás



El circuito correspondiente a este contador es más sencillo que el del contador incremental, puesto que basta con un único sumador que haga las funciones de restador para decrementar 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 53. Circuito secuencial correspondiente al contador hacia atrás



5.

La EFSM del velocímetro tiene un punto en común con el contador: la velocidad es, de hecho, la cuenta de veces que $c = 1$ hasta que $s = 1$. Sin embargo, en este caso no hay que contar a cada ciclo y, por lo tanto, harán falta 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 hace falta una variable, V, para almacenarla. Esta variable se actualizará cada vez que pase un segundo, es decir, cada vez que $s = 1$.

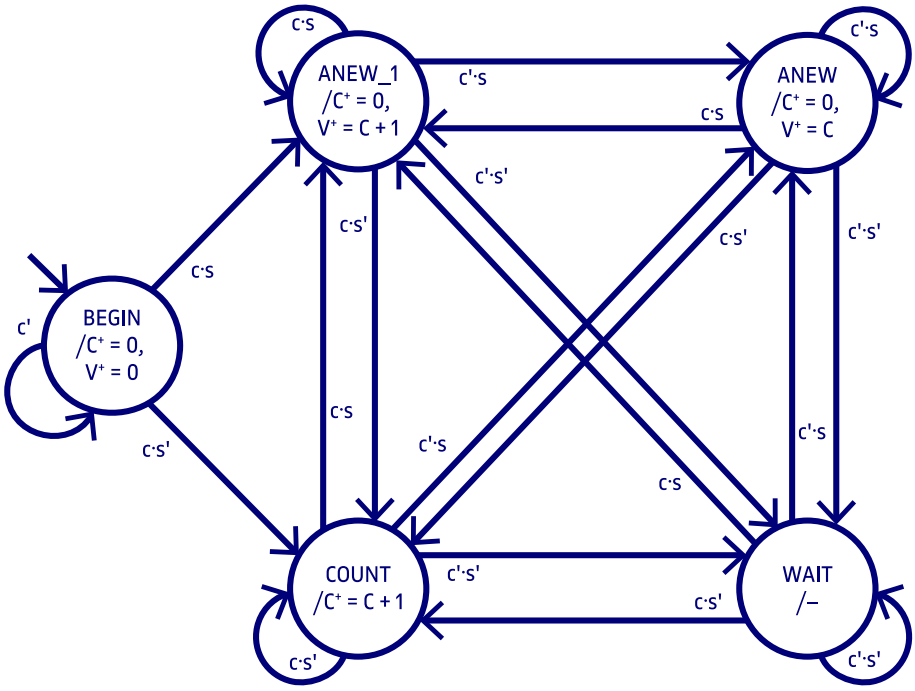
Para su cálculo, hará falta una variable que almacene el número de centímetros que se han detectado, C.

Si se da el caso de que, en un mismo periodo de reloj, s y c están en 1, habrá que tenerlo en cuenta: el centímetro se puede tener en cuenta 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 en 1, que quiere decir 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, en la figura 54.

A pesar de que tiene una apariencia complicada, las transiciones son sencillas, puesto que hay que pensar en las acciones asociadas a cada estado. Así, en COUNT se va cada vez que hay que contar un centímetro sin que haya pasado un segundo ($c \cdot s'$), en WAIT, cada vez que las entradas están en cero ($c' \cdot s'$), en ANEW, cada vez que se acaba una cuenta y se tiene que actualizar V ($s \cdot c'$) sin considerar un centímetro simultáneo, lo cual se hace en ANEW_1.

Figura 54. Grafo de estados del velocímetro



El grafo de estados se puede simplificar si se supone que, desde el principio, se hace un *reset* al circuito. Siendo así, los registros *C* y *V* también se pondrían a cero y ya no haría falta el estado BEGIN. El estado inicial tendría que ser, en este caso, el de espera, WAIT.

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

6.

La tabla de verdad se puede construir a partir de la representación de la ASM correspondiente a la figura 21.

Estado actual		**Entradas**						**Estado siguiente**	
Identificación	S_{4-0}	***d***	***t***	***r***	***s***	***e***	***h***	**Identificación**	S^+_{4-0}
IDLE	00001	0	x	x	x	x	x	IDLE	00001
IDLE	00001	1	0	x	x	x	x	UP	00010
IDLE	00001	1	x	0	x	x	x	UP	00010
IDLE	00001	1	1	1	x	x	x	MORE	00100
MORE	00100	x	x	x	x	x	x	HEAT	01000
HEAT	01000	x	x	x	0	0	x	HEAT	01000
HEAT	01000	x	x	0	0	1	0	UP	00010
HEAT	01000	x	x	0	0	1	1	KEEP	10000
HEAT	01000	x	x	1	0	1	x	MORE	00100
KEEP	10000	x	x	x	0	x	x	KEEP	10000
KEEP	10000	x	x	x	1	x	x	UP	00010

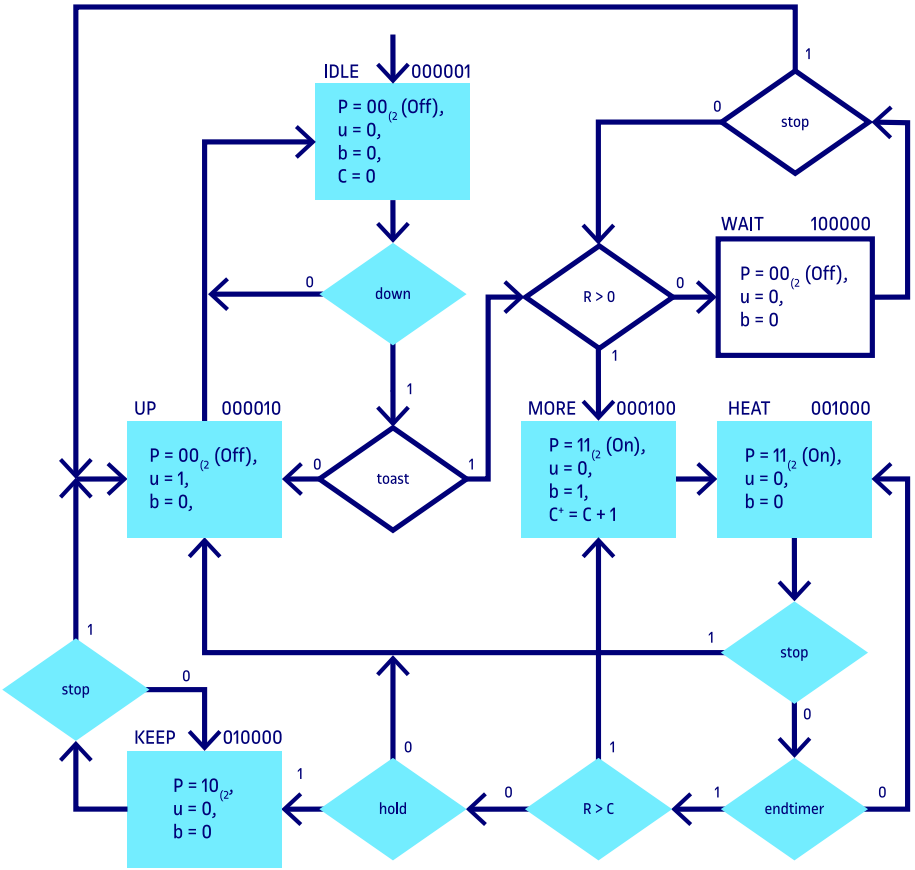
Se puede comprobar que hay muchos *don't-cares* que pueden ayudar a obtener expresiones mínimas para las funciones de cálculo del estado siguiente, pero también que son difíciles de manipular de manera manual.

7.

La modificación implica dividir en dos la decisión sobre *toast* & ($R > 0$), de forma que primero se pregunte si se ha detectado tostada y, en caso afirmativo, se compruebe si $R > 0$ para pasar a MORE. En caso de que R sea 0, se pasará a un nuevo estado (WAIT) en que se esperará que se pulse el botón de parada (*stop*) o se gire la rueda a una posición diferente de 0.

Las acciones asociadas a WAIT son las de mantener los elementos calefactores apagados sin hacer saltar la rebanada de pan hacia arriba ni activar el temporizador.

Figura 55. ASM del controlador de una tostadora con estado de espera



8.

En cuanto a los multiplexores ligados 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 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 hagan falta. Ahora bien, no todos los recursos de cálculo se usan en todos los periodos de reloj y, por lo tanto, los resultados que generan en esos periodos de tiempo no importan, del mismo modo que tampoco importa qué entradas tengan. Así pues, 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

Estado	<<	+	CMP
S_3	$P << 1$	x	$C = n$

En el circuito de la figura 24, todos los *don't-cares* aparecían como 0 en las entradas de los multiplexores. Sin embargo, con vistas a la optimización se tienen que 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, habría bastante con un multiplexor de dos entradas. Algo similar ocurre con el decalador. Sin embargo, en este caso resulta más conveniente usar dos decaladores que tener un multiplexor (los decaladores son mucho más simples que los multiplexores, puesto que no necesitan puertas lógicas).

En el caso de los registros, hay que tener en cuenta que la elección entre mantener el valor y cargar uno de nuevo se puede hacer con la señal de carga (*load*) correspondiente, lo que ayuda a reducir la orden de los multiplexores en su entrada. Es conveniente tener una tabla similar que ayude a visualizar las optimizaciones que se pueden hacer, 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 tiene que hacer una carga en el estado S_0 y, por lo tanto, basta usar S_0 conectado a la señal correspondiente del registro asociado. Algo similar se puede hacer con *C*, si la puesta a cero se hace aprovechando el *reset* del registro. En ese caso, se haría un *reset* en el estado S_0 y una carga al estado S_1. Con *P* es necesario usar 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 hace con *reset*.

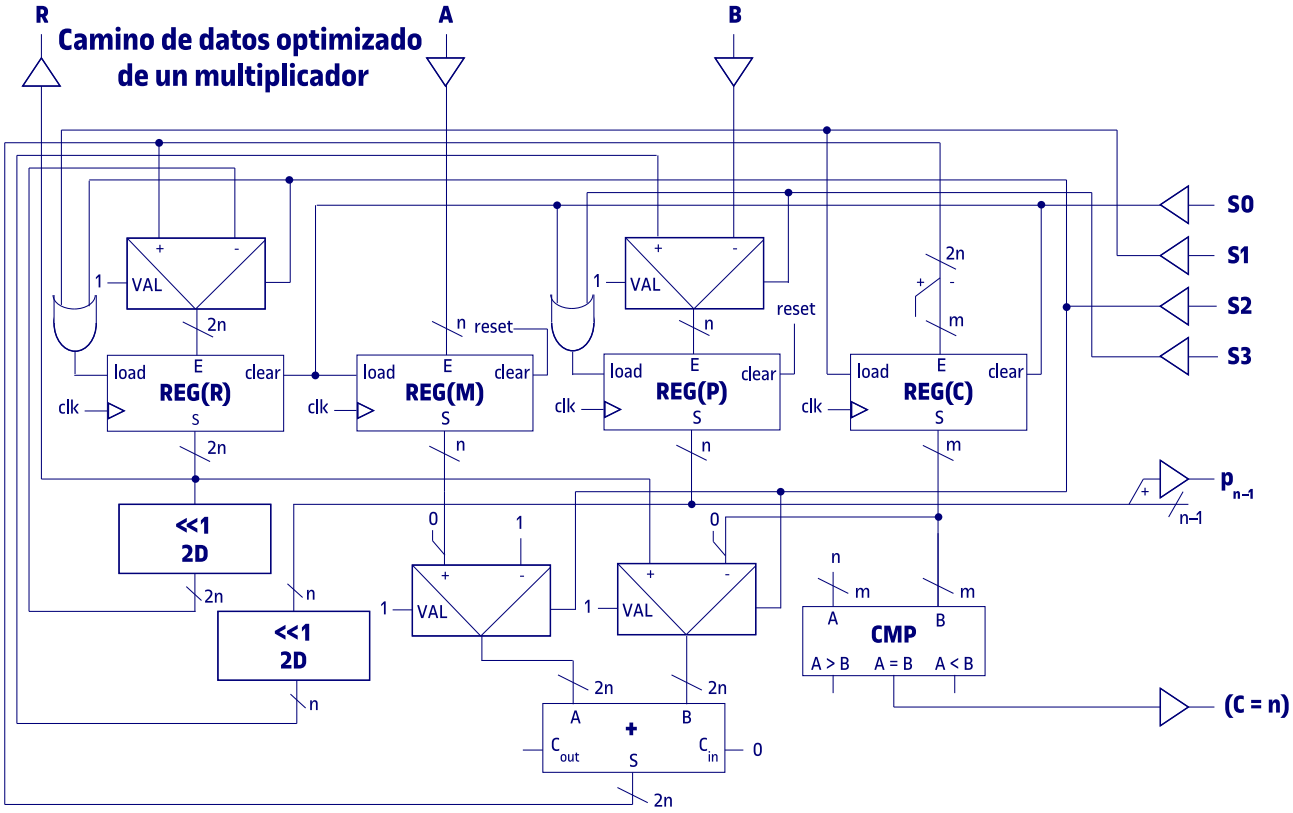
Finalmente, hay que tener en cuenta que las funciones lógicas que generan los valores de control de los multiplexores, de los *loads* y de los *resets* suelen ser más simples con la codificación *one-hot bit* que con la binaria. De hecho, en este caso no hay mucha diferencia, pero usar la codificación *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. En la tabla, aquellas operaciones en que se carga un valor están separadas por puntos suspensivos, de forma que la activación de la señal de carga se muestra como «*P* =...» o «*R* =...» y la selección del valor a cargar como «... B», «... P << 1», «... *R* << 1» y «... *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 a continuación, en la figura 56.

Figura 56. Unidad de procesamiento optimizada de un multiplicador en serie



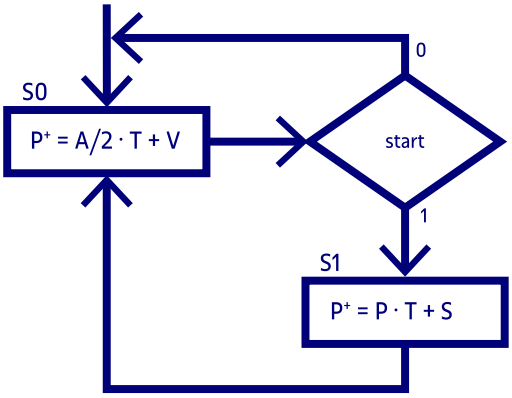
9.

Como solo se puede hacer un producto en cada etapa y en la expresión original hay tres productos, harían falta tres estados. Ahora bien, la expresión se puede reescribir como:

$P = ((T \cdot A/2) + V) \cdot T + S$

De este modo, solo hay dos productos. El primero se puede hacer en el mismo estado de espera de *start*. Hay que tener en cuenta que la operación $A/2$ es un simple desplazamiento a la derecha con copia del bit de signo, por si el dato fuera negativo. Con todo, la ASM queda como se muestra a continuación, en la figura 57.

Figura 57. ASM de un módulo de cálculo de $P = A \cdot T^2/2 + V \cdot T + S$



10.

Para completar la codificación se tiene que hacer la traducción de 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 hacer (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 00001 001	Fuerza que el resultado sea cero.
0Ch	JZ 04h (sub)	11 00001100	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 00001 001	
11h	JZ 04h (sub)	11 00001100	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 00001 001	
hlt,15h	JZ 15h (hlt)	11 010101	En espera, hasta que se le haga *reset*.

11.

Con el repertorio de instrucciones del Femtoproc, no es posible copiar el valor de un registro a otro de diferente de manera directa. Ahora bien, se puede aprovechar la instrucción NOT para poder guardar en un registro el complemento del contenido de otro registro fuente. Después, basta con hacer una operación NOT del primer registro, de forma que este registro sea fuente y destino a la vez.

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 00001 001	Fuerza que el resultado sea cero.
0Ch	JZ 04h (sub)	11 00001100	Vuelve a hacer otra resta.
pos,0Dh	NOT R3, R4	10 011 100	$R3^{+} = -R4 = R3 - R2$
0Eh	NOT R3, R3	10 011 011	$R3^{+} = -R3 = -(-R4)$
0Fh	JZ 12h (end)	11 010010	
10h	AND R0, R1	01 00001 001	
...	...	...	...

12.

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 pues, si se quieren desplazar los bits del registro 3 un bit a la izquierda basta con hacer `ADD R3, R3`.

13.

Siguiendo el patrón de estos materiales y del ejemplo que se da, hace falta 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 hace falta ningún selector para controlar ningún multiplexor adicional.

Ahora bien, los registros que cargan el dato proveniente del *BusW* cargan el resultado de hacer alguna operación con la ALU. Esta operación puede tener dos operandos, uno de los cuales es el registro *MB* y el otro puede ser *PC*, *A* o *X*. Hace falta, pues, 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, pues, hacen falta 10 microórdenes, una de las cuales (*selBusR*) es de dos bits y la 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, consiguientemente, se podría implementar con secuenciador de la figura 33.

14.

En este caso, se trataría de una instrucción de un único byte, puesto que no hay ningún operando en memoria. Así pues, solo habría que leer la instrucción, comprobar que sea de incremento de *X* (suponemos que su símbolo es INX), hacer la operación y pasar a la instrucción siguiente. Esto último se puede hacer simultáneamente con la lectura del código de operación, tal como se ha 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 y 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
		...	

15.

El *pipe* se irá llenando hasta que se acabe la ejecución de la segunda instrucción. Habiendo acabado de la fase de ejecución de la operación (EO), el *pipe* se «vacía» de forma que las fases ya iniciadas de las instrucciones número 3, 4 y 5 no se continúan. Por lo tanto, el *pipeline* tendrá una latencia de 4 periodos de reloj más hasta ejecutar la operación de una nueva instrucción.

Figura 58. *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
Periodo	1	2	3	4	5	6	7	8	9

Las *X* del *pipeline* indican que los valores de los registros a la salida de las etapas correspondientes no importan, puesto que se tienen que calcular de nuevo.

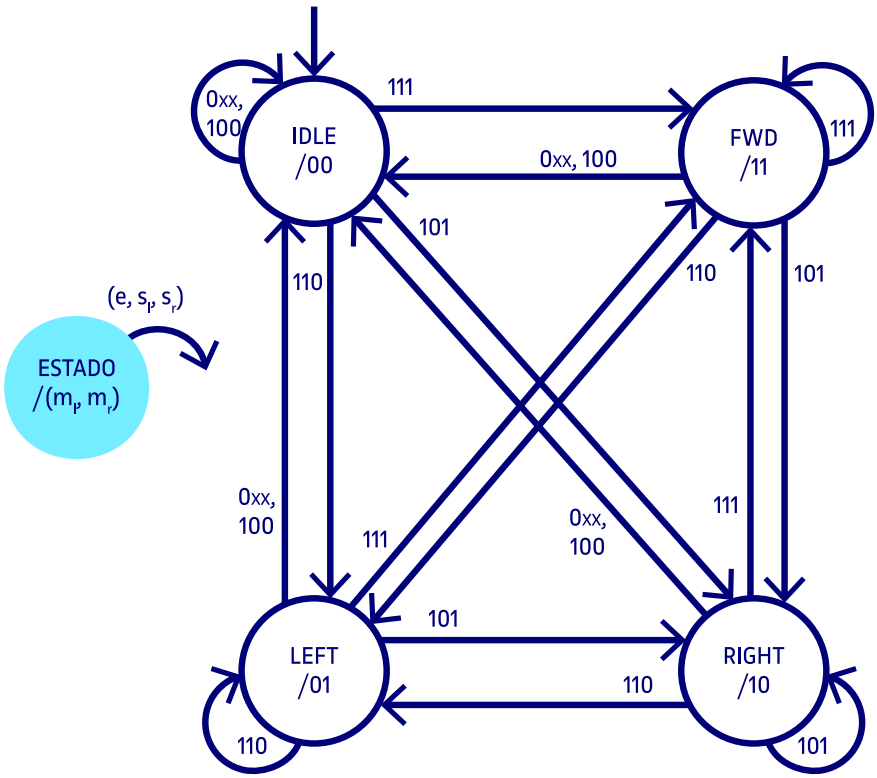
Ejercicios de autoevaluación

1. Para hacer 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 tiene que hacer nada. En ese estado, la salida tiene que ser $(m_l, m_r) = (0, 0)$, puesto que el vehículo tiene que permanecer inmóvil. Tanto si la entrada e es cero como si es 1 pero no se detecta línea en ninguno de los sensores que dan las señales de entrada, el robot se tiene que mantener en ese estado. De hecho, desde cualquier otro estado que tenga esta máquina de estados se tiene que 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 ir adelante (FWD) si se detecta línea en los dos sensores o en 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 modo parecido, cada uno de estos tres estados debe tener un arco que vaya a los otros dos, y uno de reentrante para el caso correspondiente en el mismo estado.

El dibujo del grafo correspondiente es un poco complicado porque todos los nodos están conectados entre sí (figura 59).

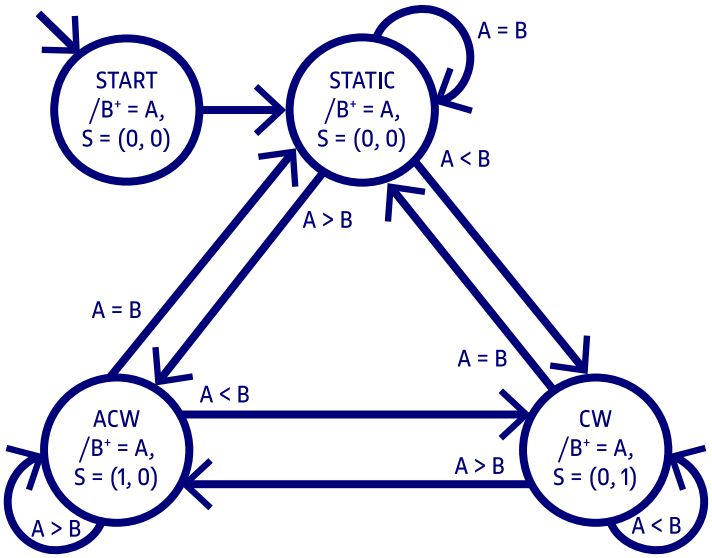
Figura 59. Grafo de estados del controlador de un seguidor de líneas



2. El enunciado del problema define tanto el comportamiento que ha seguir el detector como las entradas y salidas que tiene, así como su codificación. Ahora bien, hay que tener en cuenta que las EFSM pueden usar variables. De hecho, dado que la detección del sentido de giro se hace comparando la posición angular actual (A) con la anterior, la EFSM correspondiente debe tener una que la almacene, que se denominará B. Así pues, si $A > B$ el eje gira en sentido contrario a las agujas del reloj y la salida tiene que 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 tiene que ser (0, 0) y se debe 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 el anterior, de este estado se tiene que pasar a otro en que se pueda hacer una segunda lectura. Así, se puede pasar de START a STATIC con la misma salida. Desde STATIC se tendrá que 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, en la figura 60.

Figura 60. 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 falta codificar la variable *B* y los estados: *B* debe tener el mismo formato que *A*, es decir, debe ser un número natural de 3 bits, y los estados se codifican según la numeración binaria, de forma que START = 00, puesto que es el estado al que se debe pasar en caso de *reset*. Como se puede ver en la tabla de transiciones siguiente, se ha hecho que los estados ACW y CW tengan una codificación igual a la de las salidas *S* correspondientes:

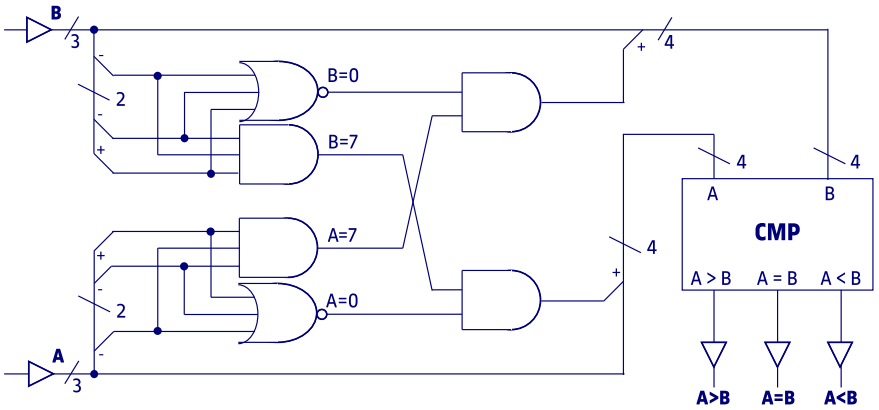
Estado actual			**Entradas**			**Estado siguiente**		
Identificación	q_1	q_0	$A > B$	$A = B$	$A < B$	**Identificación**	q_1^+	q_0^+
START	0	0	x	x	x	STATIC	1	1
CW	0	1	0	0	1	CW	0	1
CW	0	1	0	1	0	STATIC	1	1
CW	0	1	1	0	0	ACW	1	0
ACW	1	0	0	0	1	CW	0	1
ACW	1	0	0	1	0	STATIC	1	1
ACW	1	0	1	0	0	ACW	1	0
STATIC	1	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 ligadas a cada estado, la asignación $B^+ = A$ se tiene que hacer en todos ellos y, consiguientemente, se hace de manera independiente al estado en que se esté. La función de salida S es muy sencilla:

$S = (s_1, s_0) = (\text{ACW}, \text{CW}) = (q_1 \cdot q_0', q_1' \cdot q_0)$

Para la implementación, hay que tener en cuenta algo sobre el comparador: debe ser un «comparador circular», es decir, debe 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 pase, una de las opciones es usar un comparador convencional de 4 bits en que el cuarto bit sea cero excepto en el caso en que el número correspondiente sea 00001 y el otro sea 111. De este modo, 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 ese comparador circular se muestra en la figura 61.

Figura 61. 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 se ha visto. La unidad de control se ocupa de decidir el estado siguiente del EFSM y las acciones asociadas a cada estado. En este caso, tiene que 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.

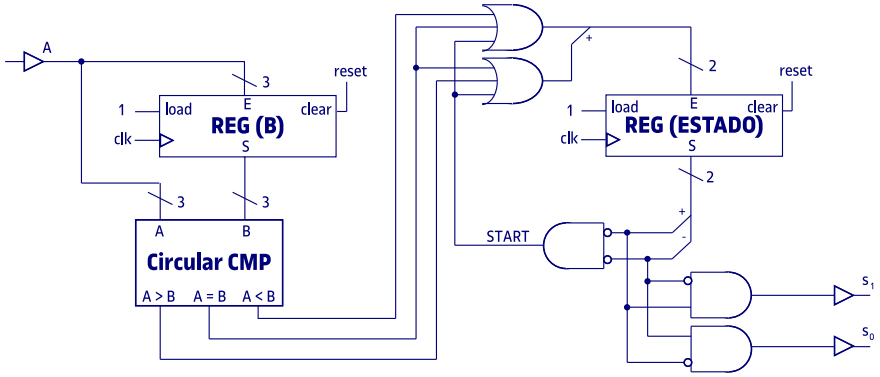
$$q_1^+ = q_1' \cdot q_0' + (A = B) + (A > B)$$

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

En la unidad de procesamiento hay que efectuar la operación de carga de la variable B y hay que generar las señales ($A < B$), ($A = B$) y ($A > B$), lo cual se hace con un comparador circular.

El circuito de tipo FSMD que materializa la EFSM del detector de sentido de giro es el que se muestra a continuación, en la figura 62.

Figura 62. Circuito secuencial correspondiente al detector de sentido de giro



3. Para el diseño de la EFSM hay que tener presente que habrá, como mínimo, un estado inicial anterior a que el sensor de nivel haya enviado algún dato y en que no se tendrá que activar la alarma. En otras palabras, mientras no llegue esta información, el controlador estará apagado o en *off*. Hay que permanecer en ese estado, que se puede denominar OFF, mientras no llegue un bit de START por la entrada s, es decir, mientras ($s = 0$) o, lo que es el mismo, s'.

En el momento en que ($s = 1$), hay que hacer 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 del primer bit, READF. En ese estado, se inicializa la variable que contiene el valor del nivel del depósito L con 3 bits en cero y el bit más significativo del valor del porcentaje (DATA3 de la figura 44) puesto como valor de unidad, es decir al hacer:

$$L^+ = 000s$$

Los otros pasos son similares, puesto que consisten en decalar a la izquierda (multiplicar por 2) el valor de L y sumar el bit correspondiente:

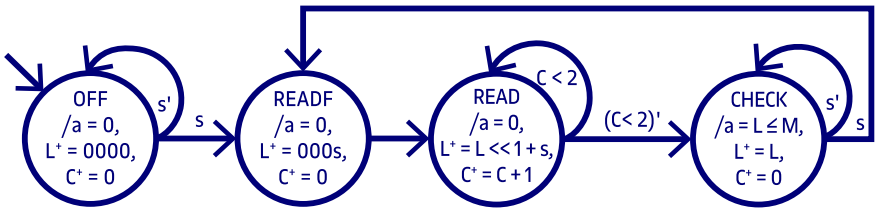
$$L^+ = (L << 1) + s$$

Pero, como la operación es diferente, se hace en el estado de lectura READ. Como hay que leer tres bits más, hace falta un contador para saber cuántos ciclos de lectura se han hecho. Este contador se inicializa en el estado READF y se incrementa en una unidad por cada ciclo que se está en READ. De este modo, después de 3 ciclos, el bit más significativo ya está en la posición de más a la izquierda y el menos significativo, en la posición de las unidades. Hay que tener en cuenta que el contador C vale 0 en el primer ciclo y, por lo tanto, la condición para mantenerse en el estado READ es ($C < 2$). Así, la máquina estará 3 ciclos en READ: C = 0, 1 y 2. Cuando llegue a 2, pasará a CHECK.

En el estado CHECK, la salida de la alarma a es el resultado de la comparación entre L y M. La EFSM se debe mantener en ese estado hasta que se reciba un nuevo dato, es decir, hasta

que $s = 1$. Como hay cuatro bits de parada en cero (STOP3,..., STOP0), no es posible perder ninguna lectura o hacer una lectura parcial de las series de bits que provienen del sensor.

Figura 63. EFSM del controlador de la alarma de aviso de nivel mínimo



Para la implementación en un circuito, con una codificación numérica de los estados, CHECK = 11 y, por lo tanto, la señal de salida a sería:

$a = s_1 \cdot s_0 \cdot (L \leq M)$

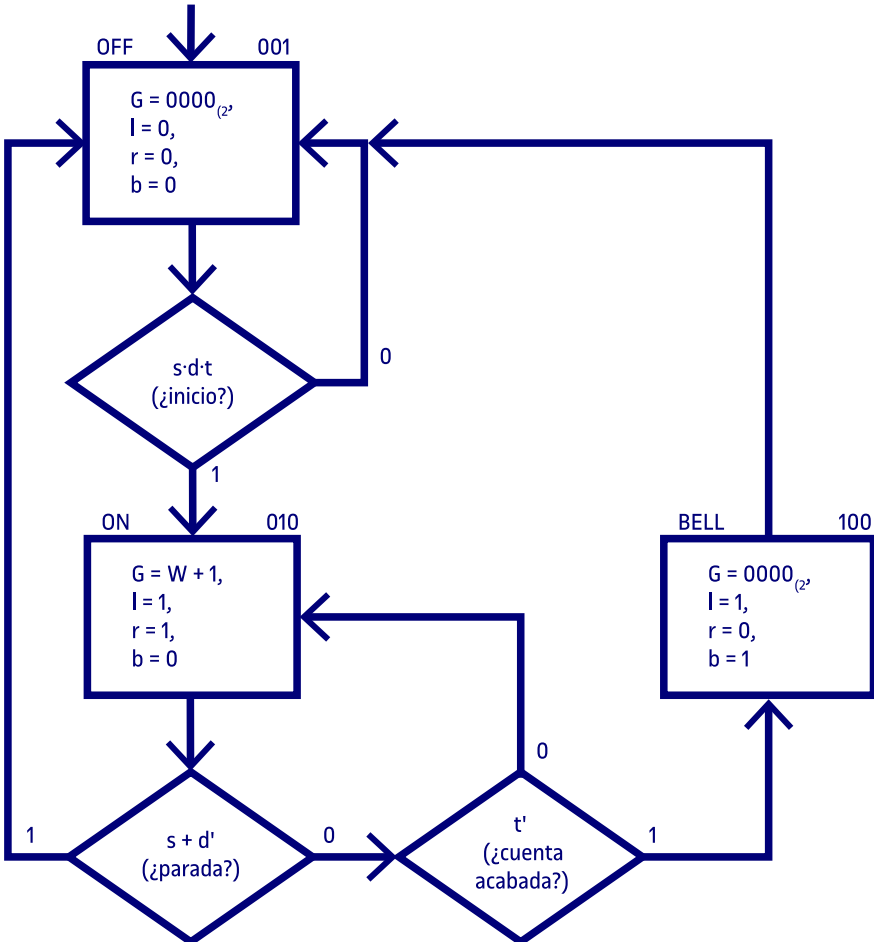
En la unidad de procesamiento tendría que haber un registro para L, otro para C y los módulos combinacionales necesarios para hacer las operaciones de decalaje ($L << 1$), suma y comparación de L, y para incrementar o poner a cero el contador C.

La unidad de control debe tener dos salidas para seleccionar qué valor se asigna a cada registro: sL, de dos bits, y sC, de un bit.

4. De manera parecida al diseño de las máquinas de estados, el procedimiento para diseñar las ASM incluye la elaboración del grafo correspondiente a partir de un estado inicial. En ese caso, el horno tiene que estar apagado (estado OFF). Tal como se indica en el enunciado, el horno tendrá que estar en OFF hasta el momento en que el usuario pulse el botón de inicio/parada ($s = 1$). En ese 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 (estado ON).

En el estado ON, hay que indicar al generador de microondas la potencia con que tiene que trabajar, que será el valor de la potencia indicada por el usuario (de 0 a 3) incrementada en una unidad. El horno debe permanecer en este estado hasta que el temporizador se acabe o se interrumpa su funcionamiento bien abriendo la puerta o bien pulsando el botón de inicio/parada. En los últimos casos, se tiene que pasar directamente al estado OFF. En el supuesto de que se acabe normalmente el funcionamiento porque el temporizador indique la finalización del periodo de tiempo ($t = 0$), se tiene que pasar a un 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 salidas en este estado es relativamente irrelevante: solo serán activos durante un ciclo de reloj. La ASM correspondiente se puede ver en la figura 64.

Figura 64. ASM del controlador de un horno de microondas

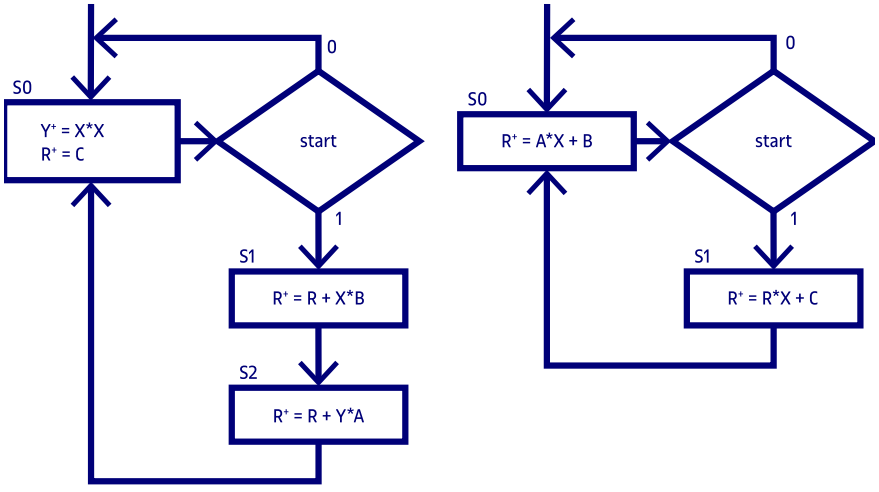


En el esquema anterior se pueden ver las expresiones lógicas asociadas a cada nodo de decisión. En ese caso, no se usa ninguna variable y, consiguientemente, la relación entre nodos de decisión y de estado es flexible.

En cuanto a la implementación de este controlador, tiene una unidad de procesamiento muy simple, dado que solo hay un cálculo ($W + 1$) y una alternativa de valores para G: $000010_{(2}$ o $W + 1$. Así pues, 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 debería servir para controlar el valor en la salida G.

5. Como solo se puede hacer un producto por ciclo y la expresión original tiene tres multiplicaciones, incluida la de X por X, hacen falta tres estados:

Figura 65. ASM de un módulo de cálculo de $A{\cdot}X^2 + B{\cdot}X + C$, directo (izquierdo) y optimizado (derecha)



Se puede reducir un estado si se transforma la expresión a $(A{\cdot}X + B){\cdot}X + C$, tal como se ve en el diagrama de la parte derecha de la figura 65.

El número de bits de R, teniendo en cuenta que los datos de entrada son de 4 bits, será el necesario para representar el valor máximo que dé la expresión:

$$15{\cdot}15{\cdot}15 + 15{\cdot}15 + 15 = 50865 < 65536 = 2^{16}$$

Por lo tanto, harán falta 16 bits para R y 8 para Y, puesto que $15{\cdot}15 = 225 < 256 = 2^8$.

6. La implementación del circuito se hace 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 la ASM: los nodos de procesamiento son biestables y los de decisión, de multiplexores.

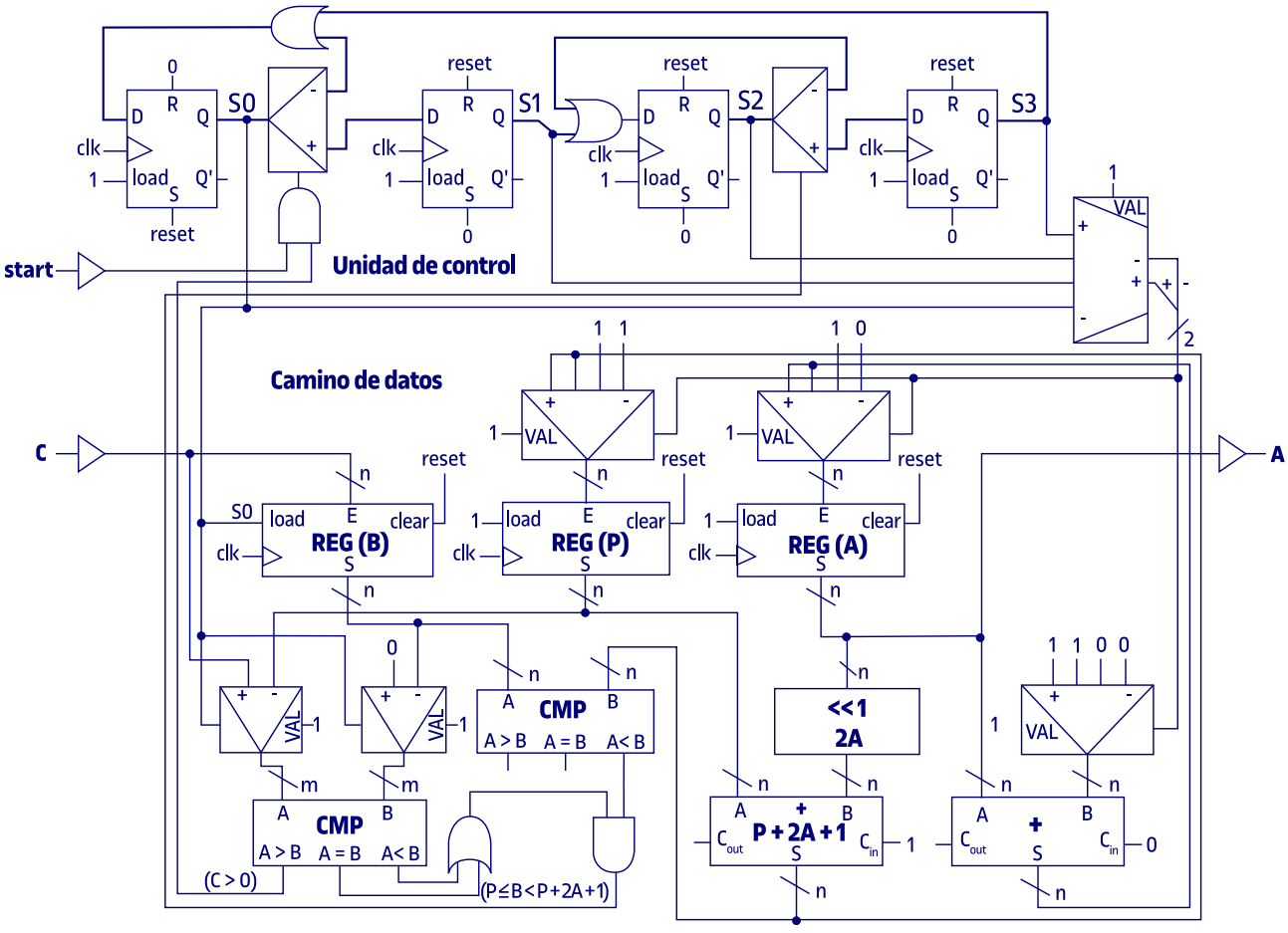
La unidad de control de la figura siguiente usa un único demultiplexor para la condición *start* y $(C > 0)$ porque se tienen que cumplir las dos para pasar a S_1. A cambio, hay una puerta AND para hacer el producto lógico entre *start* y $(C > 0)$.

El camino de datos sigue la arquitectura con multiplexores controlados por el número de estado a la entrada de los recursos, ya sean de memoria (registros) o de cálculo. En el caso de la solución que se presenta en la figura siguiente, hay algunas optimizaciones para reducir las dimensiones del esquema del circuito:

- Se ha eliminado el multiplexor a la entrada del registro B porque este registro solo carga un valor en S_0 (el contenido de la señal de entrada C).
- Se ha reducido la orden de los multiplexores a la entrada del comparador que se ocupa de calcular $(C > 0)$ y $(P \le B)$, porque la primera comparación solo se hace en S_0.
- Se han eliminado los multiplexores del cálculo de $P + 2A + 1$ y de $((P \le B)$ AND $(B < P + 2A + 1))$, porque solo se aprovechan en S_2.

Hay todavía otras optimizaciones posibles que se dejan como ampliación en el ejercicio.

Figura 66. Circuito para el cálculo de la raíz cuadrada entera



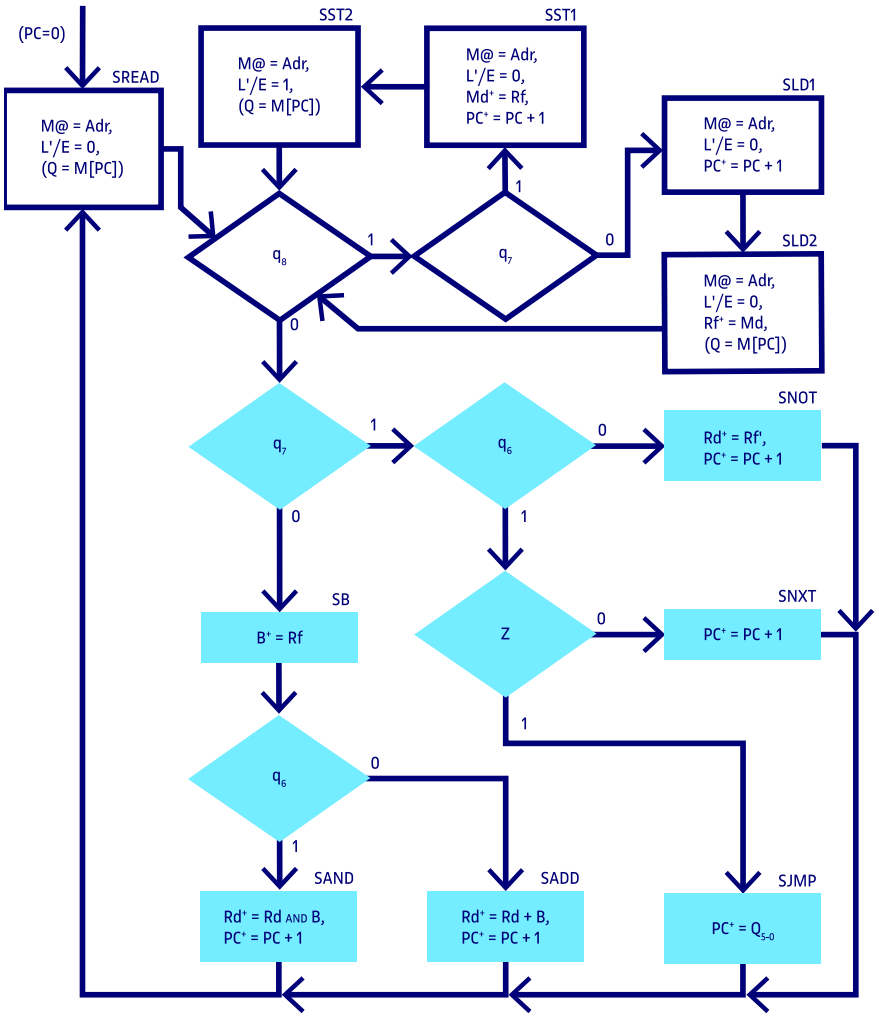
7. Tal como se muestra en la tabla que se da en el enunciado, el algoritmo de interpretación no cambia para las instrucciones que hay en el repertorio del Femtoproc. Por lo tanto, solo hay que añadirle los nodos de procesamiento ligados a las nuevas instrucciones, así como los de decisión para llegar, en particular, el que distingue entre el repertorio existente y las nuevas instrucciones, que consulta 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 siguiente. Si es 1, se pasa a interpretar las nuevas instrucciones.

El bit q_7 identifica si la instrucción es de lectura (LOAD) o de escritura (STORE) de un dato en memoria. Si se trata de una lectura, hace falta 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. Se aprovecha este mismo estado (SLD1) para actualizar PC. Dado que la memoria necesita un ciclo de reloj para poder obtener el dato de la posición de memoria dada, hace falta un estado adicional (SLD2) en 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 tiene que 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.

Si se ejecuta una instrucción STORE, $Q_{8-7} = 11_{(2}$, se tiene que pasar a un primer estado (SST1) en que se indique a la memoria a qué dirección se tiene que acceder ($M@ = Adr$) para escribir y qué contenido hay que ponerle ($Md^+ = Rf$). Este estado, además, sirve para calcular la dirección de la instrucción siguiente ($PC = PC + 1$) a la memoria del programa. El nodo siguiente (SST2) proporciona la señal de salida ($L'/E = 1$) y la espera necesaria para que la memoria haga la escritura.

Hay que tener presente que, aunque no se haya puesto, todos los estados sombreados incluyen $M@ = Adr$ y $L'/E = 0$.

Figura 67. 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, se podría añadir un segundo registro de 8 bits, igualmente co-

nectado 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 tendría que 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 tendría que implementar un ciclo de ejecución de instrucciones en qué 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, puesto 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.

algoritmo *m* Método para resolver un problema que se especifica como una secuencia de pasos finita. Cada paso consiste tanto en una decisión como en un proceso que, a su vez, puede ser otro algoritmo.

ALU *Arithmetic logic unido* (véase *unidad aritmeticològica*).

arquitectura *f* Modelo de construcción que describe los componentes (recursos) de un sistema y las relaciones que deben tener.

arquitectura de Harvard *f* Arquitectura de un procesador en 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.

arquitectura de Von Neumann *f* Arquitectura de un procesador en que el programa es almacenado en la memoria.

ASM *Algorithmic state-machine* (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 identificar 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 una multitud 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 al otro. Se denomina así porque los datos se procesan de forma que siguen un determinado camino desde las entradas hasta convertirse en resultados en las salidas.

computador *m* Máquina capaz de procesar información de manera automática, para lo cual dispone, al menos, de un procesador y de varios dispositivos de entrada y salida de información.

controlador *m* Circuito que puede actuar sobre otro elemento (habitualmente, otro circuito) mediante cambios en las señales de salida. Normalmente, estos cambios son en función del estado interno y de las señales de entrada, que suelen incluir información sobre el elemento que controlan.

core *m* (véase *núcleo*).

CPU *Central processing unit* (véase *unidad central de procesamiento*).

diagrama de flujo *m* Esquema gráfico con varios elementos que representan las diversas sucesiones de acciones posibles de un determinado algoritmo.

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 mismo computador (teclado, pantalla táctil, ratón, etc.) y una de 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 permite introducir y extraer información. Normalmente, es una memoria secundaria a la del procesador y presenta una arquitectura con dos partes: una de externa, como por ejemplo unidades de disco duro u óptico o como los conectores de los módulos para memorias USB,

y una de interna, que incluye el controlador del dispositivo o el módulo de entrada/salida correspondiente, pero que está 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 al mismo computador (pantalla, impresora, altavoz, etc.), a pesar de que hay una de más interna (el controlador del dispositivo o el módulo de salida correspondiente), aunque periférica al procesador.

DMA *Direct memory access* (véase *acceso directo a memoria*).

DSP *Digital-signal processor* (véase *procesador digital de señal*).

EFSM *Extended finite-state machine* (véase *máquina de estados finitos extendidos*).

estructura *f* Conjunto de componentes de un sistema y su relación entre ellos.

FSM *Finite-state machine* (véase *máquina de estados finitos*).

FSMD *Finite-state machine with data-path* (véase *máquina de estados finitos con camino de datos*).

GPU *Graphics processing unit* (véase *unidad de procesamiento de gráficos*).

Harvard (arquitectura de) *f* véase *arquitectura de Harvard*.

ISA *Instruction-set architecture* (véase *arquitectura del repertorio de instrucciones*).

lenguaje máquina *n* Lenguaje binario inteligible para una máquina. Normalmente, consiste en una serie de palabras binarias que codifican las instrucciones de un programa.

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.

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.

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.

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.

MCU *Micro-controller unit* (véase *microcontrolador*).

memoria *f* véase *memoria caché*.

memoria caché *f* Memoria interpuesta en el canal de comunicación de dos componentes para hacer más efectiva la transmisión de datos. Normalmente, hay entre la CPU y la memoria principal y entre el procesador y los 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.

memoria principal *f* Memoria del procesador.

microarquitectura *f* Arquitectura de un determinado procesador.

microcontrolador *n* Procesador con una microarquitectura específica para ejecutar programas de control. Normalmente, se trata de procesadores con módulos de entrada/salida de datos varios y numerosos.

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.

periférico *m* véase *dispositivo periférico*.

***pipeline* (tubería)** *f* Encadenamiento en cascada de varios segmentos de un mismo cálculo para poderlo hacer 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.

programa *m* Conjunto de acciones ejecutadas en secuencia.

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) lógicos.

unidad central de procesamiento *f* 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.

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 representa el funcionamiento 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.

unidad de procesamiento de gráficos *f* DSP adaptado al procesamiento de gráficos. Normalmente, con microarquitectura paralela.

unidad operacional *f* véase *unidad de proceso*.

variable *f* Elemento de los modelos de máquinas de estados que almacena información complementaria a los estados. Normalmente, hay una variable por dato no directamente relacionado 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.; Parilla Roure, L. (2003). *Sistemas digitales*. Madrid: McGraw-Hill.

Morris Mano, M. (2003). *Diseño digital*. Madrid: Pearson-Education.

Ribas i Xirgo, L. (200001). *Pràctiques de fonaments de computadors*. Bellaterra (Cerdanyola del Vallès): Servei de publicacions de la UAB.

Ribas i Xirgo, L. (2010). «Yet Another Simple Processor (YASP) for Introductory Courses on Computer Architecture». *IEEE Transactions on Industrial Electronics* (vol. 57, nº 10, pág. 3317-3323). Piscataway, Nueva Jersey: IEEE.

Ribas i Xirgo, L. y otros (2010). «La robótica como elemento motivador para un proyecto de asignatura en Fundamentos de Computadores». *JENUI 2010. Actas de las XVI Jornadas de Enseñanza Universitaria de la Informática*. (pág. 171-178). Santiago de Compostela: Universidad de Santiago de Compostela.

Roth, J., C. H. (2004). *Fundamentos de diseño lógico*. Madrid: Thomson-Paraninfo.