# Práctica

## Presentación

Por fin llegamos al final del curso. Esta práctica servirá para sintetizar todos los conocimientos del curso y ampliarlos con el diseño de circuitos más complejos. En esta práctica se os pedirá el diseño de un circuito a partir de un grafo de estados y el diseño de una máquina de estados en concreto. Para la primera parte, tendréis que aplicar los conocimientos que habéis adquirido en este curso, como por ejemplo el de los mapas de Karnaugh o el de los sistemas de representación. Por lo tanto, se recomienda que repaséis los anteriores módulos de los materiales.

## Competencias

Conocer la organización general de un computador como circuito digital y conocer los elementos distintivos de la arquitectura de Von Neumann.

## Objetivos

- Conocer varios modelos de máquinas de estados y las arquitecturas de controlador con camino de datos.
- Haber adquirido una experiencia básica en la elección del modelo de máquina de estados más adecuado para la resolución de un problema concreto.
- Ser capaz de diseñar circuitos secuenciales a partir de grafos de transiciones de estados.

## Recursos

Los recursos que se recomienda usar por esta PEC son los siguientes:

| | |
|---|---|
| **Básicos:** | El módulo 5 de los materiales. |
| **Complementarios:** | VerilCIRC, VerilCHART y el Wiki de la asignatura. |

## Criterios de valoración

- Las respuestas se tienen que razonar. Las que no lo estén no recibirán puntuación.
- La valoración de cada apartado está indicada en los enunciados correspondientes.

## Formato y fecha de entrega

- Para dudas y aclaraciones sobre el enunciado, dirigíos al consultor responsable de vuestra aula.
- Hay que entregar la solución en un fichero PDF usando una de las plantillas entregadas conjuntamente con este enunciado.
- Se tiene que entregar a través de la aplicación de **Entrega y registro de AC** del apartado Evaluación de vuestra aula.
- La fecha límite de entrega es el **2 de junio** (a las 24 horas).

## Solución

### PRIMERA PARTE [60%]

La EFSM siguiente describe el comportamiento del controlador de un sistema de control de estabilidad simplificado para un coche:

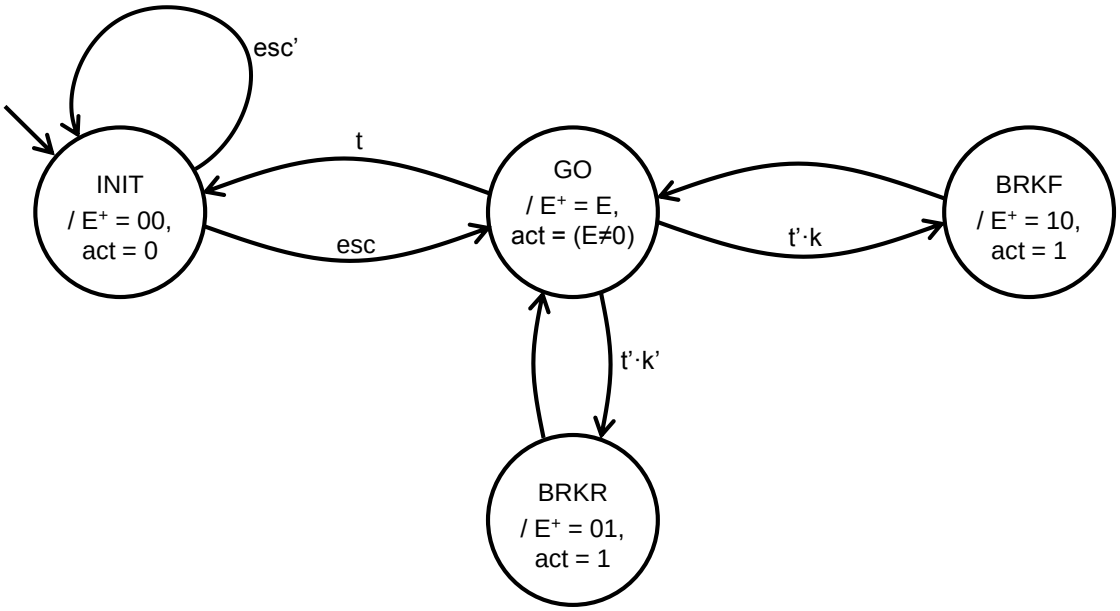


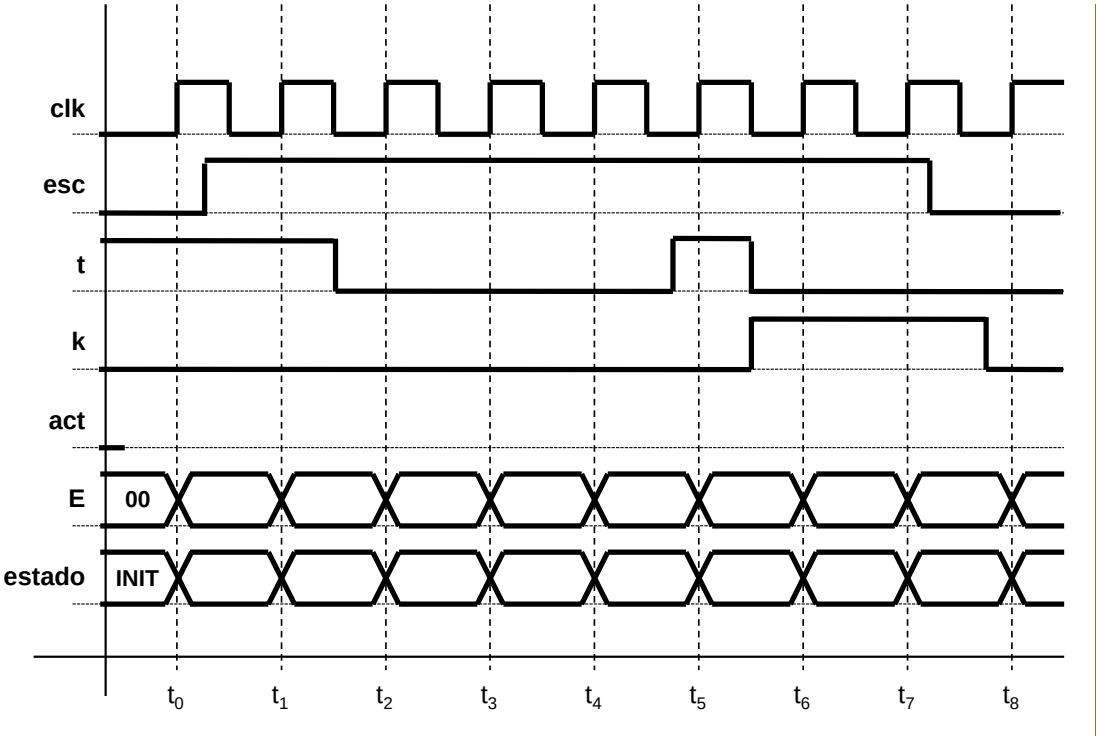

Fig. 1. EFSM de un control de estabilidad simplificado.

Los arcos sin etiqueta del grafo indican que la transición se hace en todos los casos.

Se pide:

*a*) [15%] A partir del grafo de la EFSM de la fig. 1 completar, en el cronograma siguiente, la evolución de la señal *act*, la evolución de los valores que toma *E* y el estado en que se encuentra la máquina en cada ciclo de reloj.

**Nota:** Tenéis el ejercicio disponible en VerilChart.

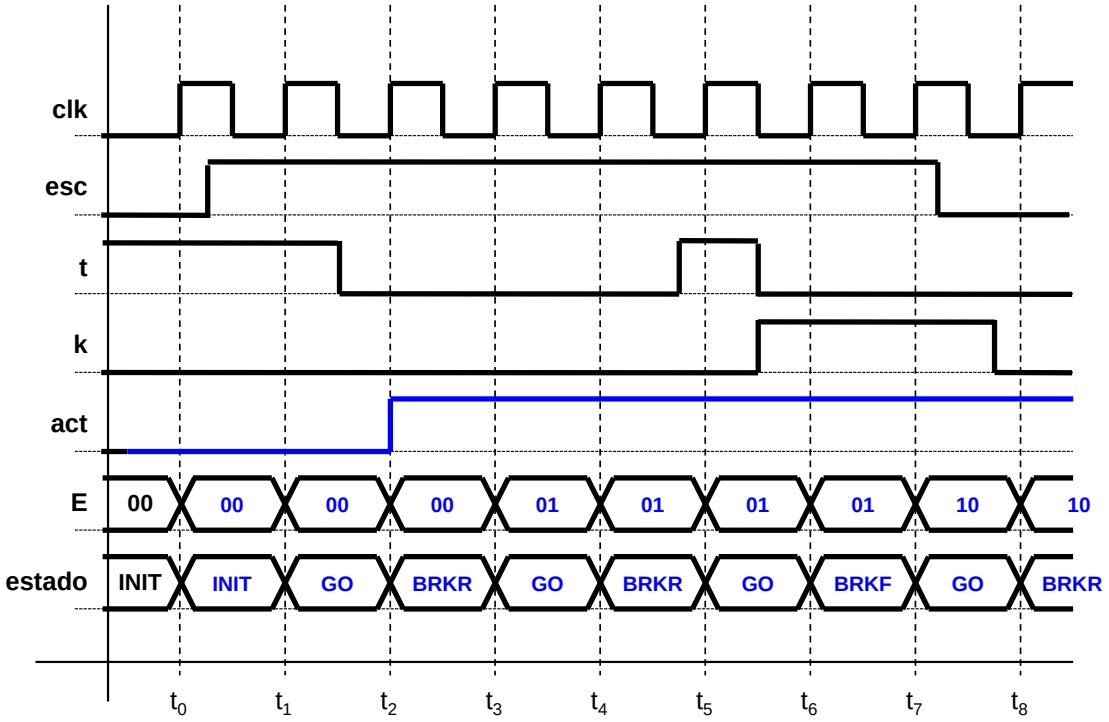



Para rellenar este cronograma, primero se determina la secuencia de estados a partir del estado anterior y de las entradas, empezando por el estado inicial, INIT, y los valores iniciales de las entradas. (En este caso, no hay que calcular ningún otro valor o series de valores porque no hay ninguna entrada que dependa de la variable *E*.)

Una vez hecho esto, se calculan la serie de valores de *E* en función de la operación que se haga para calcular $E^+$ en el ciclo anterior. Así pues, el valor de *E* en el ciclo $t_0$–$t_1$ es 00 porque, en INIT se pone $E^+$ a 00. Lo mismo pasa en el ciclo siguiente, en el que la máquina ya se encuentra en GO. En el ciclo $t_2$–$t_3$, *E* es 00 porque en GO se mantiene el contenido con $E^+=E$.

Finalmente, el valor de la señal *act* se puede determinar directamente a partir del valor que se le asigna en cada estado porque es una señal que depende, combinacionalmente, del estado actual y de la condición ($E\neq0$) en el estado actual.

Siguiendo este método, el cronograma queda de la manera siguiente:




*b*) [15%] Obtener las tablas de transiciones y de salidas de la EFSM de la fig. 1, así como las codificaciones binarias de los estados y las salidas correspondientes.

Del grafo se deduce que hay 3 señales de entrada a la FSM de control: *esc*, *t* y *k*, y salidas para determinar los valores de: *E* y *act*. Así, las tablas de transiciones y salidas que se obtienen a partir del grafo son:

| Estado actual | Entradas | Estado siguiente |
|---|---|---|
| INIT | esc' | INIT |
| INIT | esc | GO |
| GO | t | INIT |
| GO | t' ·k' | BRKR |
| GO | t' ·k | BRKF |
| BRKR | X | GO |
| BRKF | X | GO |

| Estado | Salidas |
|---|---|
| INIT | $E^+$= 00, act = 0 |
| GO | $E^+$= E, act = (E≠0) |
| BRKR | $E^+$= 01, act = 1 |
| BRKF | $E^+$= 10, act = 1 |

Para la codificación binaria, hay que asignar a cada estado un código binario individual. En este caso, se sigue el criterio más habitual, que es el de la numeración binaria, desde el 0 para el estado inicial hasta el número de estados que haya menos 1: 00 para INIT, 01 para GO, 10 para BRKR y 11 para BRKF.

En el caso de la señal *act* no hay que codificar en binario nada más, puesto que es una función combinacional que depende del código del estado y del valor de la condición ($E \neq 0$).

Para la señal *E* hace falta un selector para elegir el valor siguiente de la variable correspondiente: *s_E*. Este selector tiene que elegir entre 00, 01, 10 y mantener el valor, es decir, tiene que ser de dos bits. Así pues, la codificación binaria del selector de operación puede ser la siguiente:

| Operación | Selector s_E | Efecto sobre la variable |
|---|---|---|
| $E^+ = 00$ | 00 | Poner a 00 |
| $E^+ = 01$ | 01 | Poner a 01 |
| $E^+ = 10$ | 10 | Poner a 10 |
| $E^+ = E$ | 11 | Mantener el valor de E |

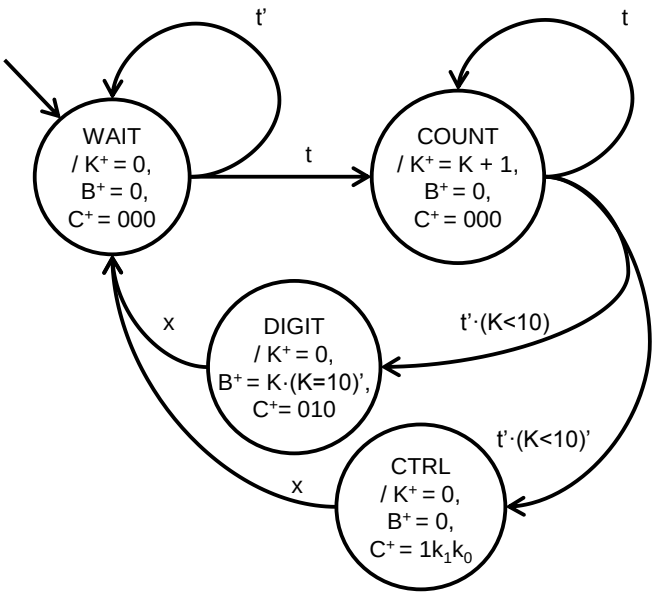
Finalmente, las tablas de transición y de salidas con la codificación binaria correspondiente son:

| Estado actual | | Entradas | | | Estado$^+$ |
|---|---|---|---|---|---|
| Símbolo | $q_1q_0$ | esc | t | k | $q_1^+q_0^+$ |
| INIT | 00 | 0 | x | x | 00 |
| INIT | 00 | 1 | x | x | 01 |
| GO | 01 | x | 1 | x | 00 |
| GO | 01 | x | 0 | 0 | 10 |
| GO | 01 | x | 0 | 1 | 11 |
| BRKR | 10 | x | x | x | 01 |
| BRKF | 11 | x | x | x | 01 |

| Estado | | Salidas | |
|---|---|---|---|
| Símbolo | $q_1q_0$ | act | s_E |
| INIT | 00 | 0 | 00 |
| GO | 01 | (E≠0) | 11 |
| BRKR | 10 | 1 | 01 |
| BRKF | 11 | 1 | 10 |

Hay que tener en cuenta que la codificación binaria de los selectores y de los estados puede ser diferente de la mostrada y que $act = q_1 + q_0 \cdot (E \neq 0)$.

En una calculadora muy simple hay, entre otros, un módulo que controla el teclado y que se ocupa de transformar cada tecla pulsada en un pulso de anchura variable que se envía a través de la señal *t*. Este pulso llega a un módulo que lo codifica en las señales *B* y *C*, donde *C* es el código de la tecla y *B* el valor en BCD en caso de que se trate de un dígito. La EFSM siguiente muestra el comportamiento de este módulo.




| #Pulsos | Significado |
|---|---|
| 0 | Ninguna tecla pulsada |
| 1 - 9 | Dígitos 1 a 9 |
| 10 | Dígito 0 |
| 11 | Introducir (↵ ) |
| 12 | Suma (+) |
| 13 | Resta (-) |
| 14 | Cancelar (C) |

Fig. 2. EFSM de un módulo codificador.

Se pide:

*c*) [10%] A partir de las tablas de transiciones y de salidas del grafo de la fig. 2 que se dan a continuación, implementar la unidad de control usando ROM y los registros, los bloques combinacionales y las puertas lógicas que se crean convenientes. Especificar y justificar las dimensiones de la/las ROM que se usen e indicar su contenido.

| Estado actual | Entradas | | Estado⁺ |
|---|---|---|---|
| | t | (K<10) | |
| WAIT | 0 | x | WAIT |
| WAIT | 1 | x | COUNT |
| COUNT | 0 | 0 | CTRL |
| COUNT | 0 | 1 | DIGIT |
| COUNT | 1 | x | COUNT |
| DIGIT | x | x | WAIT |
| CTRL | x | x | WAIT |

| Estado | | Salidas | | |
|---|---|---|---|---|
| Símbolo | $q_1q_0$ | sK | sB | sC |
| WAIT | 00 | 0 | 0 | 00 |
| COUNT | 01 | 1 | 0 | 00 |
| DIGIT | 10 | 0 | 1 | 01 |
| CTRL | 11 | 0 | 0 | 11 |

Las señales *sK*, *sB* y sC se usan como selectores que eligen cuál de los resultados se asignan a las variables *K*, *B* y *C*, respectivamente.

**Nota:** Tenéis el ejercicio disponible en VerilUOC. En esta aplicación, la señal (*K*<10) se llama *K*10.

Para implementar la unidad de control con una ROM hace falta, primero, determinar el tamaño y, después, determinar su contenido a partir de la tabla de transiciones correspondiente.

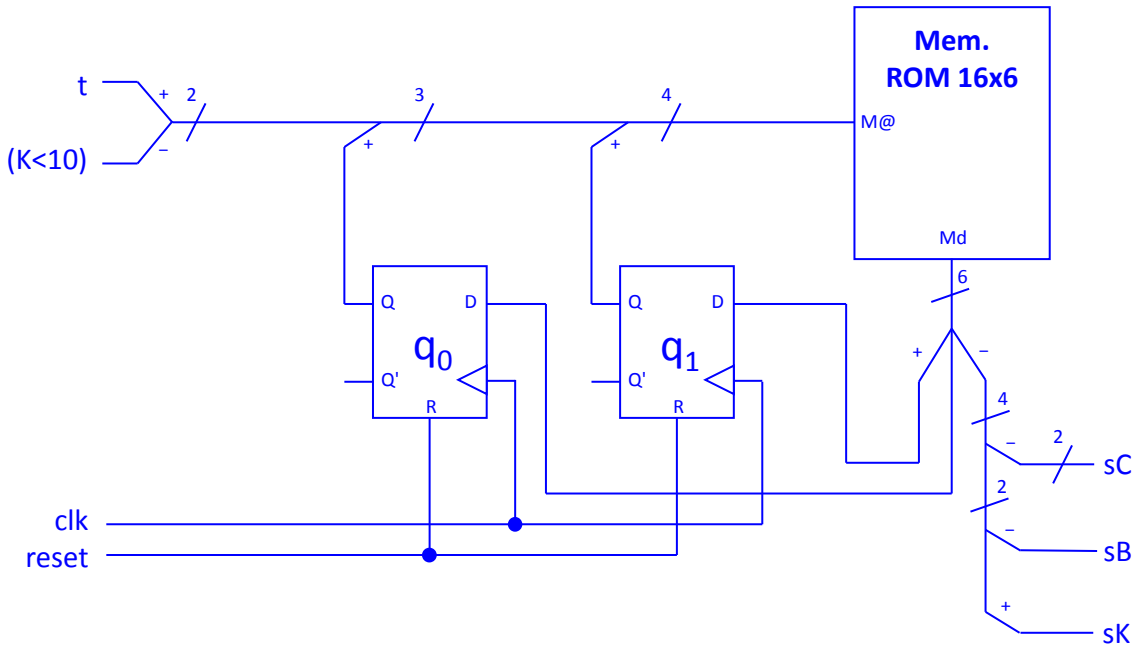
El bus de direcciones de la ROM tiene que ser de 4 bits, puesto que la dirección se forma con los bits que codifican el estado (son 2, en este caso) y los de las señales de entrada (2 más). Por lo tanto, la ROM tiene que disponer de un total de $2^4$ = 16 posiciones de memoria.

En cada posición se almacena el código del estado siguiente y el valor, en el estado actual, de cada una de las señales de salida. Por lo tanto, harán falta 2 bits para el estado y 4 bits para las salidas, cosa que hace un total de 6 bits. Así pues, la ROM tiene que ser de 16 posiciones de 6 bits.

El contenido de la ROM se puede calcular a partir de las tablas de transiciones y salidas que se dan, combinando los códigos binarios correspondientes, sin casos *don't-care*. Así, pues, la ROM tiene que tener el contenido siguiente:

| Posición de memoria | | | Contenido | | | | Codificación hexadecimal |
|---|---|---|---|---|---|---|---|
| $q_1q_0$ | t | (K<10) | $q_1^+q_0^+$ | sK | sR | sC | |
| 00 | 0 | 0 | 00 | 0 | 0 | 00 | 00h |
| 00 | 0 | 1 | 00 | 0 | 0 | 00 | 00h |
| 00 | 1 | 0 | 01 | 0 | 0 | 00 | 10h |
| 00 | 1 | 1 | 01 | 0 | 0 | 00 | 10h |
| 01 | 0 | 0 | 11 | 1 | 0 | 00 | 38h |
| 01 | 0 | 1 | 10 | 1 | 0 | 00 | 28h |
| 01 | 1 | 0 | 01 | 1 | 0 | 00 | 18h |
| 01 | 1 | 1 | 01 | 1 | 0 | 00 | 18h |
| 10 | 0 | 0 | 00 | 0 | 1 | 01 | 05h |
| 10 | 0 | 1 | 00 | 0 | 1 | 01 | 05h |
| 10 | 1 | 0 | 00 | 0 | 1 | 01 | 05h |
| 10 | 1 | 1 | 00 | 0 | 1 | 01 | 05h |
| 11 | 0 | 0 | 00 | 0 | 0 | 11 | 03h |
| 11 | 0 | 1 | 00 | 0 | 0 | 11 | 03h |
| 11 | 1 | 0 | 00 | 0 | 0 | 11 | 03h |
| 11 | 1 | 1 | 00 | 0 | 0 | 11 | 03h |

La figura siguiente muestra la implementación con una ROM de estas dimensiones. La entrada *M@* de la ROM se configura con el valor del estado actual (los 2 bits de mayor peso) y el valor de las entradas (los 2 bits de menor peso). Para guardar el estado actual se usa un registro de 2 bits.



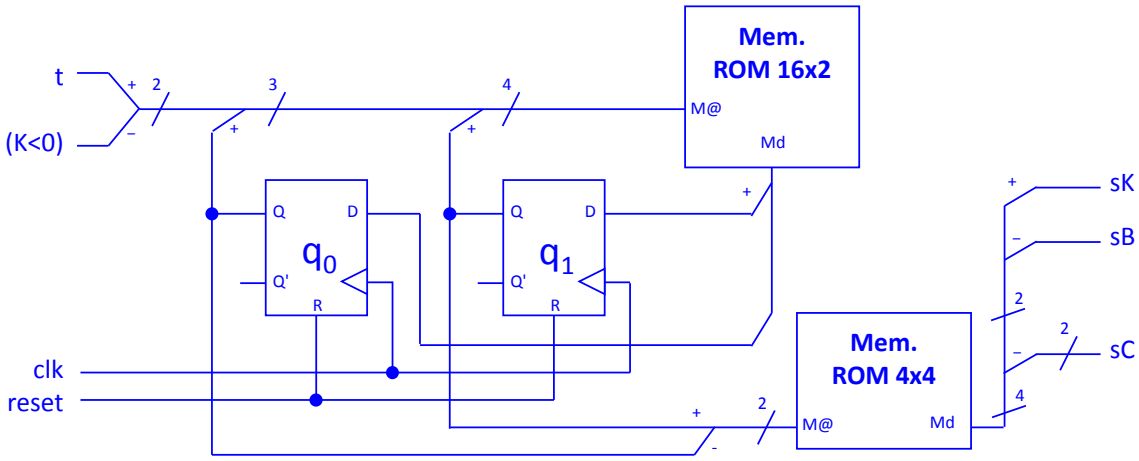

La otra opción es usando dos ROM, una para el cálculo del estado siguiente y la otra, para las salidas. En este caso, la ROM de cálculo del estado siguiente tiene el mismo número de bits de dirección que el anterior (4) pero sólo dos bits para el contenido, puesto que sólo se almacena el código del estado siguiente:

| Posición de memoria | | | Contenido | Codificación hexadecimal |
|---|---|---|---|---|
| $q_1q_0$ | t | (K<10) | $q_1^+q_0^+$ | |
| 00 | 0 | 0 | 00 | 0h |
| 00 | 0 | 1 | 00 | 0h |
| 00 | 1 | 0 | 01 | 1h |
| 00 | 1 | 1 | 01 | 1h |
| 01 | 0 | 0 | 11 | 3h |
| 01 | 0 | 1 | 10 | 2h |
| 01 | 1 | 0 | 01 | 1h |
| 01 | 1 | 1 | 01 | 1h |
| 10 | 0 | 0 | 00 | 0h |
| 10 | 0 | 1 | 00 | 0h |
| 10 | 1 | 0 | 00 | 0h |
| 10 | 1 | 1 | 00 | 0h |
| 11 | 0 | 0 | 00 | 0h |
| 11 | 0 | 1 | 00 | 0h |
| 11 | 1 | 0 | 00 | 0h |
| 11 | 1 | 1 | 00 | 0h |

La ROM del cálculo de las salidas tiene direcciones de dos bits (el código de estado) y contenidos de 4 bits, uno para *sK*, uno para *sB* y los otros dos, para *sC*:

| Posición $q_1q_0$ | Contenido sK | sB | sC | Codificación hexadecimal |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 0h |
| 01 | 1 | 0 | 00 | 8h |
| 10 | 0 | 1 | 01 | 5h |
| 11 | 0 | 0 | 11 | 3h |

Con estas dos ROM, el circuito final es:




Con esta última versión se usan menos bits de memoria ROM: de 96 (16x6) con una única ROM a 48 (16x2 + 4x4).

*d*) [20%] Implementar el circuito completo de la EFSM de la Fig. 2: Construir el camino de datos usando las puertas lógicas y los bloques necesarios e incorporar la unidad de control obtenida al apartado anterior como un módulo más. Indicar y razonar, para cada bus, su dimensión en bits teniendo en cuenta que las señales de entrada ocupan el mínimo número de bits posible según su rango, sabiendo que el rango de *K* es [0, 14],

**Nota:** Tenéis el ejercicio disponible en VerilUOC. Recordad que (*K*<10) se llama *K*10.

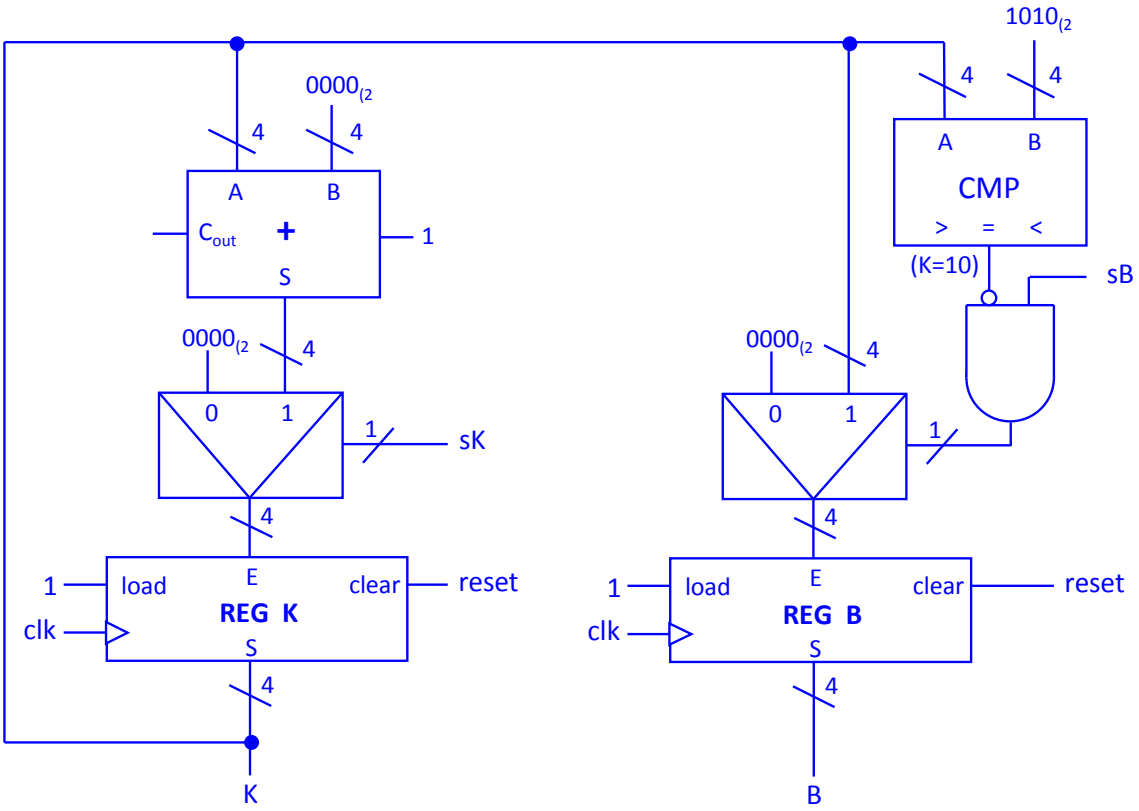
El camino de datos se tiene que ocupar, por un lado, de generar la señal (*K*<10) y, por el otro, tiene que actualizar y almacenar el valor de las variables *K*, *B* y *C*. Para esto último, tiene que usar las señales *sK*, *sB* y *sC*, respectivamente.

La generación de (*K*<10) se puede hacer a partir de un comparador de 4 bits, que son los que hacen falta para poder representar los valores del 0 al 14 en *K*.

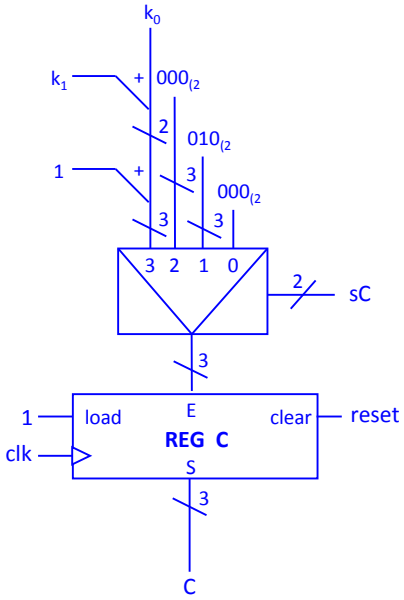
La variable *K* se pondrá a cero en el ciclo siguiente de que *sK* = 0 y se incrementará en 1 después de que *sK*=1. Esto se consigue con el circuito de la figura de más abajo.

Dado que *B* puede tener los valores de *K*, también tiene que ser de 4 bits. En este caso, cada uno de los bits de *K* se multiplica por la señal (*K*=10)', que no está generada. Por lo tanto, hace falta un comparador para generarla y un multiplexor para hacer el producto: si (*K*=10)' es 1, entonces $B^+$ será *K*, y, si no, será 0.

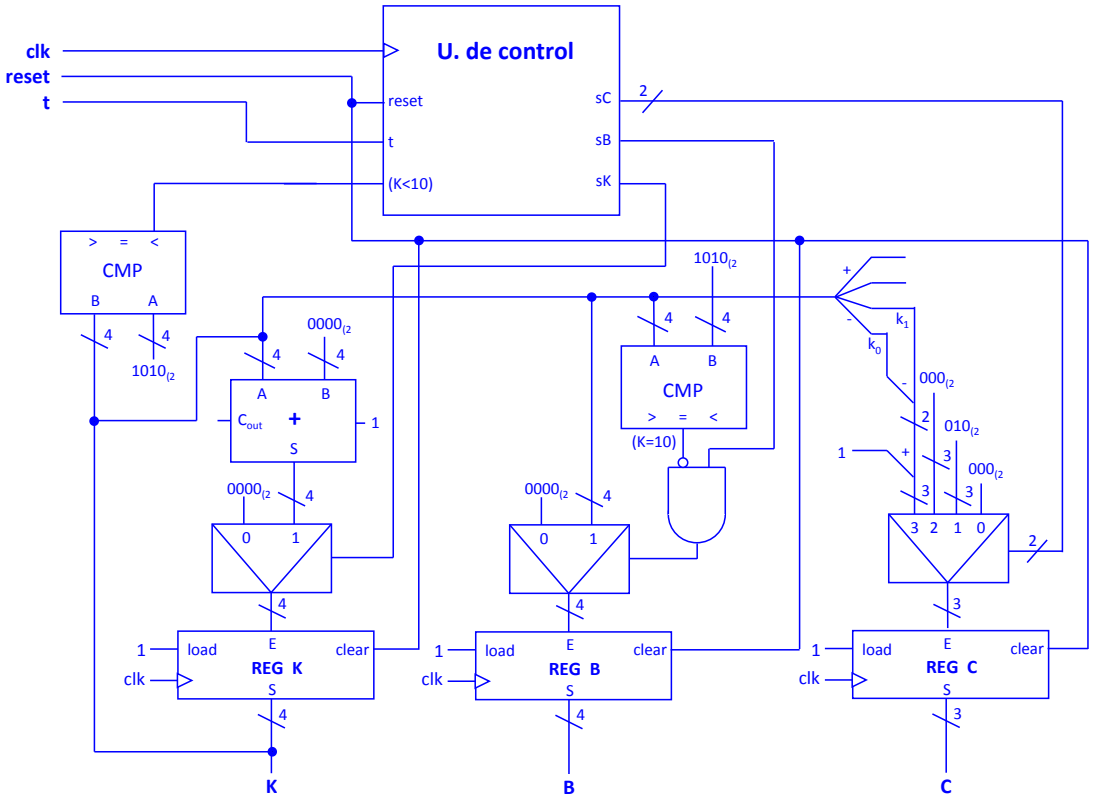
Con todo, *B* también será cero si *sB* = 0. Así pues, en lugar de usar dos multiplexores, se puede usar sólo uno, con una entrada de control combinada: $B^+$ sólo será *K* si *sB* = 1 y (*K*=10)' = 1. El circuito resultante es el que se ve a la derecha de la figura siguiente:



Para hacer el cálculo de $C^+$ sólo hace falta un multiplexor que elija entre los valores 000, 010 y $1k_1k_0$, según el valor de *sC*:




Con todos estos circuitos se monta el camino de datos necesario para la implementación de un circuito que se comporte como la EFSM de la fig. 2. El circuito completo queda tal como se muestra a continuación:



**SEGUNDA PARTE [40%]**

Se quiere diseñar un controlador que permita a un robot hacer el seguimiento de un camino marcado. El camino que tiene que seguir el robot es una línea negra, sin bifurcaciones, dibujada sobre un suelo blanco.

El robot dispone de 3 sensores S2, S1 y S0 que tienen asociados las señales $s_2$, $s_1$ y $s_0$, respectivamente. Los sensores permiten distinguir el negro del blanco. Cuando el sensor *Si se* encuentra sobre la línea negra pone la señal $s_i$ a 1. Cuando se encuentra fuera de la línea, sobre el suelo blanco, la señal $s_i$ es 0.

Los tres sensores están en la parte inferior central del robot, dirigidos hacia tierra, y dispuestos de forma transversal, con el sensor S1 en el centro, el sensor S0 a la derecha y el sensor S2 a la izquierda. La separación entre los sensores S0 y S2 es más grande que la anchura de la línea. Esto quiere decir que, con el robot correctamente situado encima de la línea, las señales $s_2$, $s_1$ y $s_0$ tienen que ser 0, 1 y 0, respectivamente.

El robot también tiene un detector de obstáculos apuntando en el sentido de la marcha que permite detectar si hay un obstáculo delante. Cuando se detecta un obstáculo, la señal *d* de salida del detector es 1, y 0 en caso contrario.

El robot se pone en funcionamiento cuando la señal *g* se pone a 1. Mientras la señal *g* sea 1 el robot se mantiene en funcionamiento y se para cuando pasa a 0.

El movimiento del robot se puede gobernar con las señales $m_l$ *y* $m_r$, que controlan los motores de las ruedas izquierda y derecha, respectivamente: Con ($m_l$, $m_r$) = (0, 0) el robot se mantiene parado, con ($m_l$, $m_r$) = (0, 1) va hacia la derecha, con ($m_l$, $m_r$) = (1, 0) va hacia la izquierda y con ($m_l$, $m_r$) = (1, 1) va recto.

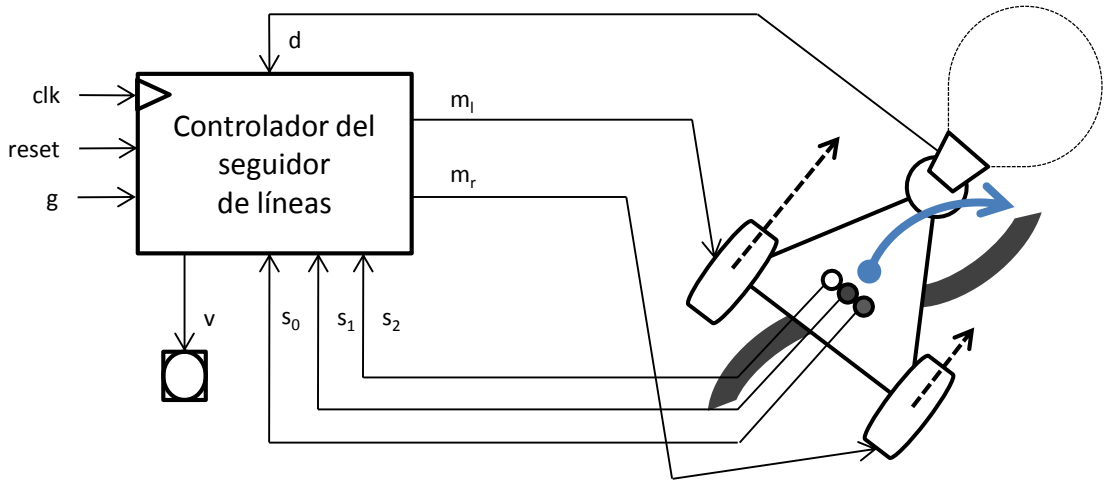



Fig. 3. Esquema de las entradas y salidas del controlador del robot.

Se espera que el controlador haga que el robot se comporte de la manera siguiente:

— Inicialmente, el robot está parado. El robot se pone en funcionamiento cuando la señal *g* pasa a 1 y se mantiene así hasta que *g* sea 0.

— Cuando el robot se pone en funcionamiento, inicia el seguimiento de la línea negra, avanzando e intentando mantenerla bajo el sensor S1 en todo momento. Se usa la

información de los sensores S0, S1 y S2 para dar las órdenes de ir recto, a la derecha o a la izquierda.

— Cuando se detecta un obstáculo delante ($d$=1), el robot se para momentáneamente. Si el obstáculo desaparece pronto (antes de que pasen $T$ ciclos de reloj), se vuelve a poner en marcha si $g$ todavía es 1. En cambio, si el obstáculo se mantiene más de $T$ ciclos de reloj, el robot se para e indica esta situación activando la señal $v$ ($v$=1), que enciende un led de color rojo.

— Si la línea negra desaparece o se detecta alguna otra situación anómala como que los tres sensores de detección de línea sean 1, el robot se para y pone $v$ a 1.

— Una vez encendida, la señal $v$ sólo se apagará cuando $g$ pase a ser cero, es decir, se espera a que $g$ pase a cero y, después, retoma el funcionamiento normal, esperando que se ponga a uno de nuevo.

Se pide diseñar la ASM del controlador del robot que hace que el robot se comporte de la forma especificada.

En el estado inicial, IDLE, se tiene que esperar que $g$=1 y, al mismo tiempo, poner las variables de las salidas a cero.

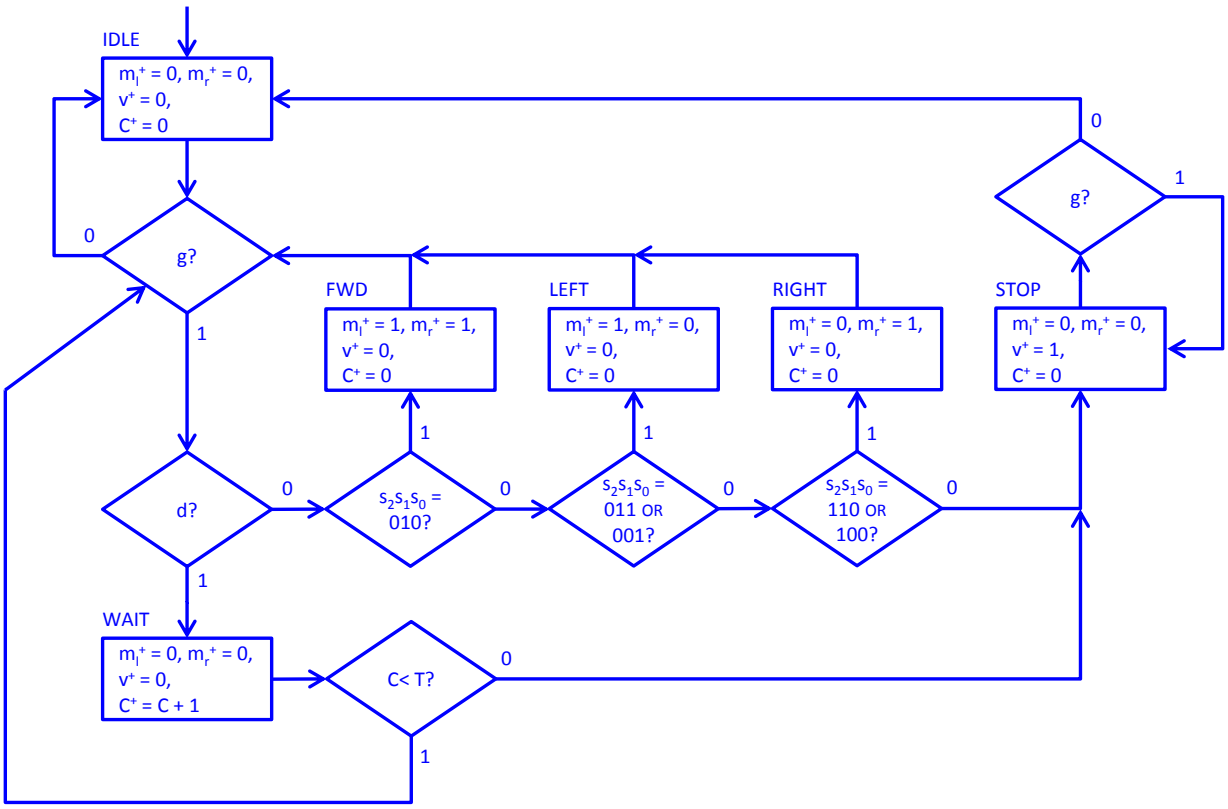
Cuando $g$=1, hay que comprobar si hay algún obstáculo. Si $d$ es 1, hay que esperar que sea 0. Si no, hay que determinar en qué situación está el robot respecto de la línea: Si está centrado, entonces tiene que moverse adelante (FWD); si está desplazado hacia la derecha, cosa que se sabe porque ($s_2$, $s_1$, $s_0$) es 011 o 001, se tiene que mover hacia la izquierda (LEFT), y, si está a la izquierda de la línea, se tiene que mover hacia la derecha (RIGHT).

Cuando tenga que esperar que $d$=0, hay que iniciar una cuenta de ciclos mientras $d$ y $g$ sean 1. Esta cuenta se hace en WAIT. De este estado sólo se sale cuando $g$=1 y $d$=0 o cuando deja de cumplirse que el contador $C$ sea más pequeño que $T$. Hay que tener en cuenta que esta última condición implica que $C$ ya superará $T$ en el estado siguiente y, por lo tanto, que hay que parar el robot. Así pues, si esto pasa, el controlador pasará al estado STOP.

En los casos que no se detecte línea, ($s_2$, $s_1$, $s_0$) = 000, que el robot esté cruzando la línea trasversalmente, ($s_2$, $s_1$, $s_0$) = 111, o que no se detecte línea en medio, pero sí a los lados ($s_2$, $s_1$, $s_0$) = 101, el robot se parará pasando también al estado STOP. En este estado se pondrá $v$=1 para indicar que no se puede continuar.

En el estado STOP se hace una espera hasta que $g$=0 y, en este caso, se retoma el funcionamiento normal a la espera de una nueva activación de $g$.

La ASM correspondiente es:




En este caso, se puede optar por no usar variables para las salidas $m_l$, $m_r$ y $v$.