Introducción a los computadores

Prácticas de laboratorio

Juan Segarra y Alejandro Valero

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Curso 2021–2022

© 2018-2022 Juan Segarra, Universidad de Zaragoza

Parcialmente basado en:

[1] N. Ayuso, F. García, L. M. Ramos y J. Segarra. Enunciados de prácticas. *Sistemas lógicos*, Ingeniería en Informática/Telecomunicación, Universidad de Zaragoza, 2003–2011.

[2] N. Ayuso, L. M. Ramos, J. Segarra, A. Valero y V. Viñals. Enunciados de prácticas. *Introducción a los computadores*, Grado en Ingeniería Informática, Universidad de Zaragoza, 2010–2017.

Esta obra está distribuida bajo una *Licencia Creative Commons BY-SA*. Para ver una copia de la licencia, visite `https://creativecommons.org/licenses/by-sa/4.0/legalcode.es`

Resumen de Licencia Creative Commons

Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)

Este es un resumen legible por humanos (y no un sustituto) de la licencia.

Usted es libre de:

Compartir — copiar y redistribuir el material en cualquier medio o formato.

Adaptar — remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

Atribución — Usted debe dar crédito de manera adecuada, proporcionar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo de cualquier forma razonable, pero no de forma que sugiera que usted o su uso tienen el apoyo del licenciante.

CompartirIgual — Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

Avisos:

No tiene que cumplir con la licencia para elementos del material en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable. No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Índice general

Desarrollo de las prácticas

En las prácticas trabajaremos con el simulador de circuitos lógicos *Digital*, desde el sistema operativo Windows. Una vez arrancado el sistema operativo, buscar y lanzar la versión ya instalada desde el menú de aplicaciones. *Digital* es software libre, que se puede descargar gratuitamente desde `https://github.com/hneemann/Digital`. Las prácticas consistirán en diseñar circuitos o experimentar con circuitos ya creados.

Preparación de la práctica

Todas las prácticas tienen un trabajo previo a realizar *antes de acudir al laboratorio*. El trabajo previo está insertado en el enunciado de cada práctica con «huecos» a rellenar. Dependiendo de cada práctica, rellenar los huecos implicará obtener o minimizar funciones, rellenar tablas de verdad, etc.

Práctica 1

Funciones booleanas, circuitos lógicos y su simulación con *Digital*

En esta práctica se aprenderá a usar *Digital* como editor/simulador de circuitos.

1.1. Configuración

Hay que lanzar *Digital* desde el menú de inicio→Programas→Prácticas→Simuladores→Digital. Al iniciar *Digital* por primera vez, es posible que no esté configurado. En ese caso, selecciona el menú Edit→Settings para configurarlo. Configura el idioma en español y verifica que están marcadas las opciones de «formas IEEE 91-1984», «ver árbol de componentes desde el inicio», «mostrar número de cables en el bus» y «consejos sobre la herramienta cable». Si te lo pide, cierra *Digital* y vuelve a ejecutarlo.

1.2. Tutorial

Al iniciar *Digital* por primera vez, a la izquierda aparece una ventana con un tutorial. En caso de que no aparezca, lanza manualmente el tutorial desde el menú Ver→Start Tutorial.

Sigue el tutorial, que muestra paso a paso cómo dibujar el circuito de la figura 1.1. Cuando llegues a la parte de simular, observa que los colores verde oscuro y verde claro representan la transmisión de un '0' y '1' lógico, respectivamente. Prueba todas las posibles combinaciones de entradas y verifica que coinciden con la tabla de verdad vista en clase para la puerta *xor*. A continuación prosigue con el tutorial para darle nombre a las entradas y salidas tal como aparece en la figura 1.1.

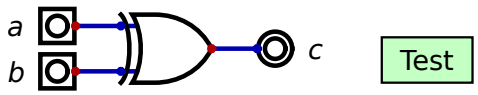
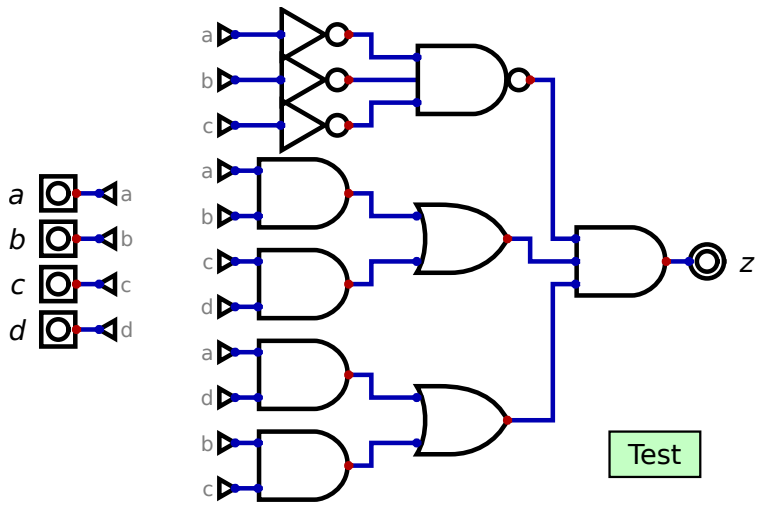


Figura 1.1: Circuito con puerta *xor* y Test.

1.3. Verificación y almacenamiento

Tras finalizar el tutorial, añade un caso de prueba para el circuito (en el menú Componentes→Varios→Test). Edita el contenido del test pulsado botón derecho sobre él y a continuación *Editar*, e introduce la tabla de verdad de la puerta *xor*. Pulsa el botón de *Ayuda* para conocer el formato. Ahora puedes validar tu circuito desde el menú Simulación→Ejecuta las pruebas. Guarda el circuito como `xor.dig` en `c:\tmp` o `c:\Temp`. Una vez guardado, puedes almacenarlo en un USB, subirlo a algún sistema de almacenamiento en la nube, enviarlo por correo electrónico, etc.



a	b	c	d	z
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Figura 1.2: Circuito a analizar y su tabla de verdad.

1.4. Implementación de funciones booleanas

Como trabajo previo a la práctica, hay que obtener varias expresiones booleanas:

1. Escribe en el siguiente hueco la expresión booleana del circuito representado en la figura 1.2, tal como has visto en clase:

   $z =$

2. A partir de la expresión rellena su tabla de verdad (figura 1.2).

3. A partir de la tabla de verdad obtén su 1ª forma canónica:

   $z =$

4. A continuación transforma la 1ª forma canónica en una expresión que sólo utilice puertas *nor* y *not*:

   $z =$

Durante la sesión de laboratorio utilizaremos *Digital* para comprobar si las expresiones anteriores son correctas y equivalentes.

5. Crea un circuito nuevo y dibuja el circuito de la figura 1.2. No conectes directamente las entradas al circuito; usa el componente *Túnel* (en *Cables*) para conectar cables a través de etiquetas, tal y como se muestra en la figura. Puedes cambiar la orientación de cada componente pulsando «r» o desde sus propiedades. También puedes cambiar el número de entradas de una puerta desde sus propiedades (clic con el botón derecho sobre ella).

6. Simula el circuito y obtén el valor de salida para algunas combinaciones de entradas. Comprueba que coinciden con tu tabla de verdad.

7. Haz que *Digital* obtenga la tabla de verdad del circuito (menú Análisis→Análisis). Comprueba que la tabla coincide con la tuya.

8. Debajo del diseño anterior, dibuja con puertas la expresión de la 1ª forma canónica (punto 3). No dibujes nuevas entradas; usa túneles para conectar las entradas a, b, c, d que ya tienes dibujadas. Para que no haya conflictos, etiqueta la nueva salida como *z2*.

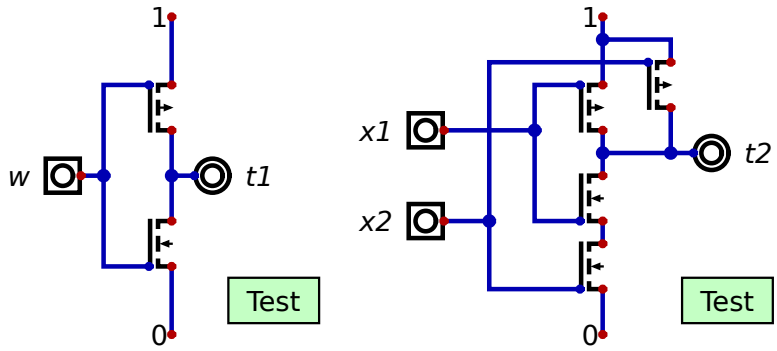


Figura 1.3: Dos puertas lógicas construidas con transistores.

9. En el mismo circuito, dibuja con puertas la expresión del punto 4. Igual que antes, usa túneles para reutilizar las entradas existentes, y etiqueta la salida como *z3*.

10. Igual que antes, haz que *Digital* muestre la tabla de verdad del circuito y comprueba que las tres salidas son iguales.

11. Como último paso, añade un test para validar el circuito.

12. Guarda el circuito con nombre `funcion.dig`.

1.5. Implementación de puertas con transistores MOSFET

En esta sección vamos a simular puertas lógicas fabricadas mediante transistores MOSFET. La figura 1.3 muestra la implementación de dos puertas lógicas mediante estos transistores. Los transistores MOSFET se comportan como interruptores controlados por el valor que se coloca en la base (señal w en el ejemplo de la izquierda). El transistor tipo N (con flecha entrando) se cierra a 0 y se abre a 1. El transistor tipo P (con flecha saliendo) se cierra a 1 y se abre a 0.

13. Crea un nuevo circuito y dibuja la figura 1.3 utilizando los componentes *FET de canal P/N* (en *Interruptores*) y *Valor constante* (en *Cables*).

14. Observa la salida ante todas las combinaciones de entradas. ¿Qué puertas son?

15. Añade un test (o varios) para validar el circuito y guárdalo con nombre `transistores.dig` para su posterior entrega.

Práctica 2

Interpretación y visualización de números enteros

En esta práctica vamos a diseñar varios circuitos que nos faciliten la interpretación y visualización de números enteros. Para ello practicaremos la minimización de funciones con mapas de Karnaugh.

2.1. Circuito conversor *BCD* → *7-segmentos*

Para visualizar dígitos decimales vamos a utilizar un *display* de 7 segmentos. Dicho componente tiene una entrada para cada uno de sus segmentos (a-g) y otra para el punto, de forma que cada segmento se ilumina cuando su entrada correspondiente vale 1. Para mostrar dígitos, los segmentos deben iluminarse tal y como se muestra en la figura 2.1. Por ejemplo, se mostrará 4 cuando los segmentos b, c, f, g valgan 1. Dado que cada display debe mostrar un único dígito decimal, es imprescindible tener dicho dígito a la entrada del circuito. Para ello aprovecharemos la codificación BCD (*Binary-coded Decimal*) vista en clase. Nuestro circuito deberá pasar de un número codificado en BCD a una salida de 7 bits para iluminar los segmentos correctos en el display.

1. Rellena la tabla 2.1. En la columna auxiliar *dígito* debes especificar el dígito decimal de la entrada BCD, es decir, un número natural entre 0 y 9. En las columnas *a-g* debes especificar los valores 0/1 que iluminen el dígito correspondiente.

2. Aplica el método de Karnaugh para obtener la expresión *mínima suma de productos* de las funciones a, b y c.

   $a =$

   $b =$

   $c =$

3. Aplica el método de Karnaugh para obtener la expresión *mínimo producto de sumas* de las funciones d, e, f y g.

   $d =$

   $e =$

   $f =$

   $g =$

En lugar de pasar manualmente las funciones a puertas, vamos a hacer que *Digital* sintetice automáticamente el circuito a partir de la tabla de verdad.

4. Selecciona Análisis→Sintetizar.

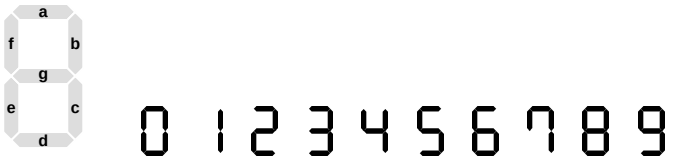


Figura 2.1: Pantalla de 7 segmentos, con su correspondiente iluminación para cada dígito.

BCD_3	BCD_2	BCD_1	BCD_0	dígito	a	b	c	d	e	f	g
0	0	0	0								
0	0	0	1								
0	0	1	0								
0	0	1	1								
0	1	0	0								
0	1	0	1								
0	1	1	0								
0	1	1	1								
1	0	0	0								
1	0	0	1								
1	0	1	0								
1	0	1	1								
1	1	0	0								
1	1	0	1								
1	1	1	0								
1	1	1	1								

Tabla 2.1: Tabla de verdad BCD→7-segmentos.

5. Desde la ventana de sintetización, selecciona Nuevo→Combinacional→4 variables.

6. Añade tantas salidas como necesites desde Editar→Añade columnas de salida.

7. Renombra cada entrada/salida con el botón derecho sobre su nombre. Los nombres de las entradas deben ser «BCD3», «BCD2», «BCD1» y «BCD0», mientras que salidas deben tomar los nombres «a», «b», «c», «d», «e», «f» y «g».

8. Introduce en cada columna de salida los valores de tu tabla con el ratón, o situándote sobre la casilla correspondiente y pulsando 1/0/x.

9. Selecciona Mapa de Karnaugh para visualizar el mapa de Karnaugh y la expresión algebraica (en forma de suma de productos) de cada salida.

10. Comprueba que cada mapa de Karnaugh coincide con el tuyo en papel y que las expresiones obtenidas coinciden o son equivalentes a las rellenadas anteriormente.

11. Selecciona Crear→Circuito.

12. En el circuito creado, sustituye las entradas de 1 bit por una única entrada de 4 bits etiquetada como «BCD». Para agrupar/dividir bits hay que usar el componente Cables→Divisor. Por ejemplo, para dividir una línea de 4 bits en cuatro líneas de 1 bit hay que poner (botón derecho) «4» para la entrada y «1,1,1,1» para la salida. Para agrupar líneas hay que poner los valores a la inversa. Pon especial atención en preservar el orden de los bits, o de lo contrario el circuito no funcionará correctamente.

13. Simula el nuevo circuito y comprueba que se cumple la tabla de verdad. Siempre es recomendable añadir un componente test para verificar la tabla. En este caso, para especificar el valor de la

entrada «BCD» de 4 bits se puede escribir el valor decimal de esos 4 bits interpretados como binario natural.

14. Guarda el circuito con el nombre `BCDto7seg.dig`.

2.2. Circuito conversor *Signo-Magnitud → Signo y BCD*

En esta sección construiremos un circuito que transforme un número codificado en Signo-Magnitud a BCD para su posterior visualización. En este caso, dado que el número a la entrada es entero, necesitaremos una salida específica S para el signo.

15. Rellena la tabla 2.2a. En la columna auxiliar *decimal* debes especificar el número entero de la entrada en decimal, es decir, interpretado de acuerdo a su codificación (Signo-magnitud en este caso). La salida S debe indicar el signo del número (1 si es negativo). Hay que tener en cuenta que en esta codificación el cero puede tener signo positivo o negativo. Las 4 salidas BCD deben indicar el único dígito del número a la entrada. Como tenemos entradas de 4 bits, para su rango de representación nunca se necesitará más de un dígito decimal. En resumen, este circuito tendrá como entradas «SM3», «SM2», «SM1» y «SM0», mientras que salidas serán «S», «BCD3», «BCD2», «BCD1» y «BCD0».

16. Minimiza las salidas usando Karnaugh y verifícalas usando *Digital*.

    $S =$

    $BCD_3 =$

    $BCD_2 =$

    $BCD_1 =$

    $BCD_0 =$

17. Sintetiza el circuito como antes, sustituyendo las entradas de 1 bit por una única entrada de 4 bits etiquetada como «SM». De manera similar, sustituye las salidas BCD de 1-bit por una única salida de 4 bits etiquetada como «BCD». Guarda circuito resultante con el nombre `SMtoBCD.dig`.

18. En un circuito nuevo (`prac2.dig`) visualizaremos el número entero $(X3, X2, X1, X0)$ que vamos a interpretar (figura 2.2). Para su interpretación como *Signo-Magnitud*, hay que usar los circuitos anteriores. Guarda el nuevo circuito como `prac2.dig` en el mismo directorio que los circuitos anteriores. Una vez guardado podrás añadir los circuitos desde Componentes→Personalizado. Si no aparecen, selecciona Componentes→Personalizado→Actualizar. Para su visualización, añade dos *displays* de 7 segmentos (Componentes→Entrada-Salida→ Más→Display de 7 segmentos), uno al lado del otro. El display de la izquierda servirá para visualizar el signo (salida S) en su segmento g. En las siguientes secciones se añadirá lo necesario para visualizar el resto de codificaciones.

19. Comprueba que los displays muestran correctamente los números a la entrada, asumiendo que son enteros codificados como Signo-Magnitud (e.g. 1010: -2, 1000: -0).

2.3. Circuito conversor *Complemento a 1 → Signo y BCD*

En esta sección construiremos un circuito que transforme un número codificado en Complemento a 1 a BCD para su visualización.

20. Rellena la tabla 2.2b.

21. Minimiza las salidas usando Karnaugh y verifícalas usando *Digital*.

    $S =$

$SM_{3...0}$	dec.	S	$BCD_{3...0}$
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

(a) Tabla SM→BCD

$Ca1_{3...0}$	dec.	S	$BCD_{3...0}$
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

(b) Tabla Ca1→BCD

$Ca2_{3...0}$	dec.	S	$BCD_{3...0}$
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

(c) Tabla Ca2→BCD

Tabla 2.2: Tablas de verdad de interpretación de codificaciones enteras.

$BCD_3 =$

$BCD_2 =$

$BCD_1 =$

$BCD_0 =$

22. Dibuja el circuito agrupando entradas/salidas y guárdalo con el nombre `Ca1toBCD.dig`.

23. En el circuito `prac2.dig`, replica lo que has hecho antes pero usando ahora `Ca1toBCD.dig` como componente (figura 2.2).

24. Comprueba que los displays muestran correctamente los números a la entrada. Por ejemplo, para Ca1, 1010 debe mostrar -5 y 1111 -0.

2.4. Circuito conversor *Complemento a 2 → Signo y BCD*

En esta sección construiremos un circuito que transforme un número codificado en Complemento a 2 a BCD para su visualización.

25. Rellena la tabla 2.2c.

26. Minimiza las salidas usando Karnaugh y verifícalas usando *Digital*.

$S =$

$BCD_3 =$

$BCD_2 =$

$BCD_1 =$

$BCD_0 =$

27. Dibuja y guarda el circuito con el nombre `Ca2toBCD.dig`.

28. En el circuito `prac2.dig`, añade los componentes necesarios para visualizar las entradas X con la nueva interpretación de las entradas Ca2→BCD (figura 2.2).

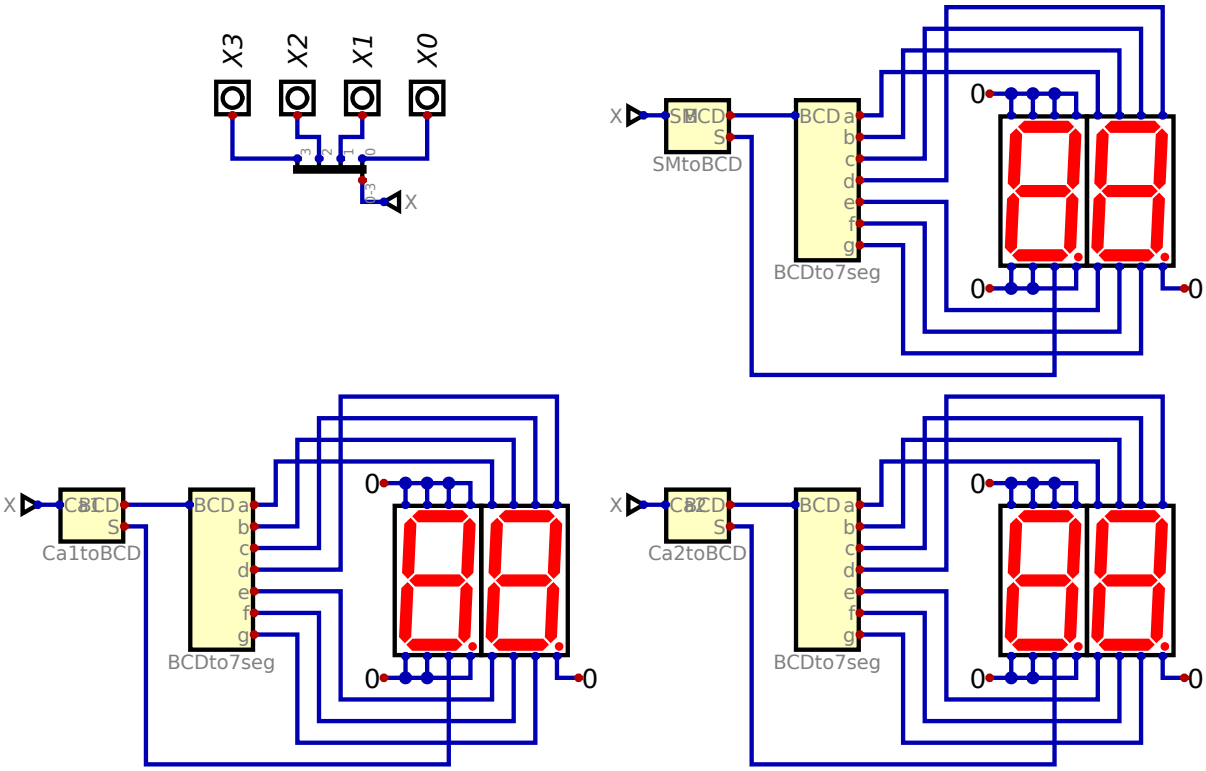


Figura 2.2: Ejemplo del circuito de visualización de enteros.

29. Comprueba que los displays muestran correctamente los números a la entrada para cada una de las codificaciones. Por ejemplo, para Ca2, 1010 debe mostrar -6 y 1000 -8.

2.5. Comparación de codificaciones

30. Para vectores de entrada con $X_3 = 0$, ¿qué muestran los displays? ¿Muestran todos lo mismo?

31. ¿Qué muestran los displays para la entrada 1111?

32. ¿Qué muestran los displays para la entrada 1000?

33. ¿Cuantos números se pueden mostrar como máximo en cada codificación? ¿A qué se deben las diferencias?

Práctica 3

Diseño de una unidad aritmético-lógica (ALU)

En esta práctica se construirá una Unidad Aritmético-Lógica (ALU) de números de 4 bits con indicador (*flag*) de cero. Para implementar la ALU necesitaremos un sumador-restador y multiplexores. Como trabajo previo, es necesario llevar a la sesión *todos* los diseños a implementar.

3.1. Implementación de un sumador-restador de 4 bits en complemento a 2

1. Crea un circuito con nombre `FA.dig`. Dibuja un sumador completo o *full adder* (dos bits A y B a sumar, más acarreo de entrada *Cin*, para generar el bit de suma S y el acarreo de salida *Cout*) usando dos puertas *and*, dos puertas *xor* y una puerta *or*, según el diseño basado en dos *half adders* visto en clase.

2. Crea un circuito con nombre `RCAS4.dig` y dibuja un sumador-restador de números de 4 bits en complemento a 2 con propagación de acarreo (*Ripple Carry Adder-Subtractor*) según el diseño visto en clase. El circuito debe tener dos entradas A, B y una salida S, todas de 4 bits. Además debe tener una entrada M de 1 bit para indicar si debe sumar ($M = 0$) o restar ($M = 1$). Por último, también debe tener una salida D de 1 bit que indique si hay desbordamiento ($D = 1$) o no ($D = 0$) asumiendo operaciones en complemento a 2.

3. Verifica el funcionamiento del circuito.

4. ¿Cuál es el resultado al realizar la resta $(-4) - 4$? ¿Y al sumar $4 + 4$?

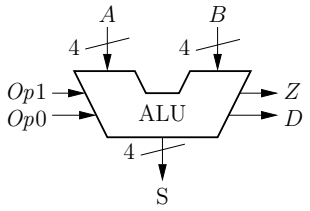
5. ¿Cuál es la ventaja de utilizar complemento a 2 en la codificación de enteros frente a otras codificaciones como signo-magnitud?

3.2. Diseño de multiplexores

6. Crea un nuevo circuito `MUX4.dig` y diseña un multiplexor 4:1 (4 entradas seleccionables $x3, x2, x1, x0$ de 1 bit, dos entradas selectoras $s1, s0$, y una salida z) mediante puertas lógicas y cableado.

7. Crea otro circuito `MUX4x4.dig` y diseña un multiplexor 4:1×4 (4 entradas seleccionables $x3, x2, x1, x0$ de 4 bits, dos entradas selectoras $s1, s0$, y una salida z de 4 bits) usando el componente `MUX4.dig` tantas veces como necesites.

8. Verifica el funcionamiento del circuito.



$Op1$	$Op0$	S
0	0	$A+B$
0	1	$A-B$
1	0	B
1	1	1

$$Z = \overline{S_3 + S_2 + S_1 + S_0}$$

$$D = \begin{cases} D_{RCAS4} & \text{si } S = A \pm B, \\ 0 & \text{en caso contrario.} \end{cases}$$

Figura 3.1: Esquema y especificaciones de la ALU a diseñar.

3.3. Diseño de la ALU

Para diseñar la ALU se necesitan los componentes anteriores. La ALU a diseñar tiene dos vectores de entrada A y B de 4 bits, y genera como resultado otro vector S de 4 bits. Para nuestra ALU, podemos ver las distintas operaciones y sus códigos de control asociados en la tabla de la figura 3.1. Dado que la ALU debe ser capaz de generar como resultado una de entre varias operaciones (especificadas mediante dos entradas de control $Op1, Op0$ de 1 bit cada una), una forma de implementarla es hacer que realice las posibles operaciones y seleccionar, con un multiplexor, cuál de los resultados hay que propagar a su salida. Además, la salida Z indicará si el resultado de la operación realizada ha sido 0 o no ($Z = 1 \leftrightarrow S = 0$) y la salida D indicará si hay desbordamiento ($D = 1$) en la operación realizada.

9. En un circuito nuevo (`ALU4.dig`) construye la ALU especificada usando los componentes anteriores, y usando un único componente `RCAS4.dig`. Enlaza los componentes a las entradas A y B, y utiliza el multiplexor `MUX4x4.dig` para generar la salida S en función de las entradas de control $Op1, Op0$.

10. Haz en papel una tabla de verdad especificando el valor deseado para M en función de las entradas $Op1, Op0$ y aplícalo al circuito. Obtén la expresión para D de forma similar. Finalmente, añade la salida Z a partir de la salida S.

11. Verifica el funcionamiento del circuito. Puedes crear un circuito nuevo en el que usar la ALU y ver el resultado, por ejemplo reutilizando los circuitos `Ca2toBCD.dig` y `BCDto7seg.dig` de la práctica anterior, tal y como se muestra en la figura 3.2. Para poder utilizar circuitos ya hechos, recuerda que es necesario que se encuentren en el mismo directorio que el circuito actual.

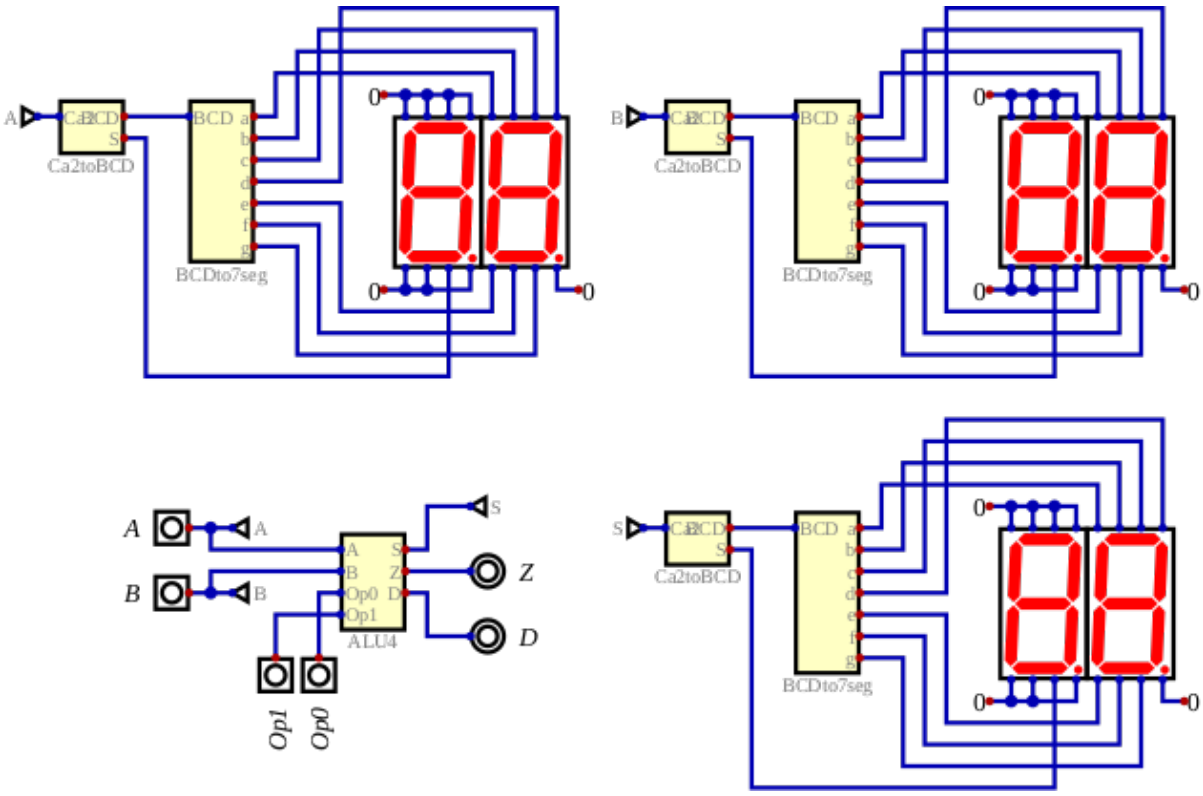


Figura 3.2: Ejemplo de circuito de verificación de la ALU.

Práctica 4

Análisis temporal y análisis de sistemas secuenciales

En esta práctica se analizarán los tiempos de propagación en circuitos combinacionales y se analizará un circuito secuencial.

4.1. Tiempos de propagación en puertas lógicas

En una puerta lógica real, construida con transistores, un cambio en el valor lógico de una entrada no se manifiesta de forma instantánea como un cambio en la salida. Tiene que transcurrir un tiempo que se conoce como *retardo de propagación*[1].

1. Asumiendo que cada puerta atravesada tiene un retardo de 1 μs, ¿cuál será el retardo (máximo) de cada una de las salidas del circuito `FA.dig` de la práctica anterior?

2. Dibuja un circuito sumador tal y como se muestra en la figura 4.1 y guárdalo como `Retardos.dig`. Como se puede observar, este circuito contiene las puertas de cuatro *full adders*.

3. Añade Sondas para A, B, S y C_{out} (Componentes→Entrada-Salida→Sonda).

Una forma de visualizar cambios en las señales con respecto al tiempo es utilizar un *cronograma*. Con el simulador funcionando, puedes visualizarlo desde el menú Simulación→Mostrar gráfico de medidas. A partir de este momento, cualquier cambio en las entradas del circuito se verá reflejado en el cronograma. Como en el circuito no hay elementos temporales, la simulación habitual asume que cada cambio incrementa el tiempo en una unidad y deja el sistema estabilizado. En este caso *no* queremos visualizar el cronograma global del circuito, sino visualizar el retardo puerta a puerta. Para ello, hay que iniciar la simulación con el modo de puerta sencilla. En este modo de simulación, los cambios en las entradas no se aplican instantaneamente, sino que se aplicarán puerta a puerta.

4. Pulsa  para iniciar la simulación en modo de puerta sencilla.

5. Cambia B a 1 y observa que las puertas afectadas por dicho cambio se han marcado con un círculo azul celeste.

6. Pulsa  para simular un paso (retardo de una puerta). Observa que las señales se han propagado desde las puertas marcadas a las siguientes (marcadas ahora con el círculo azul celeste). La salida S debe seguir valiendo 0 porque aún no ha llegado el cambio introducido en la entrada.

[1] El retardo de propagación depende del tamaño de los transistores, del número de entradas de la puerta y del número de puertas que se conectan a su salida. También depende, a veces, del sentido de la transición, siendo necesario distinguir entre tiempo de subida y tiempo de bajada. En esta asignatura asumiremos que cada puerta tiene un tiempo de retardo único, independiente de su conexionado.

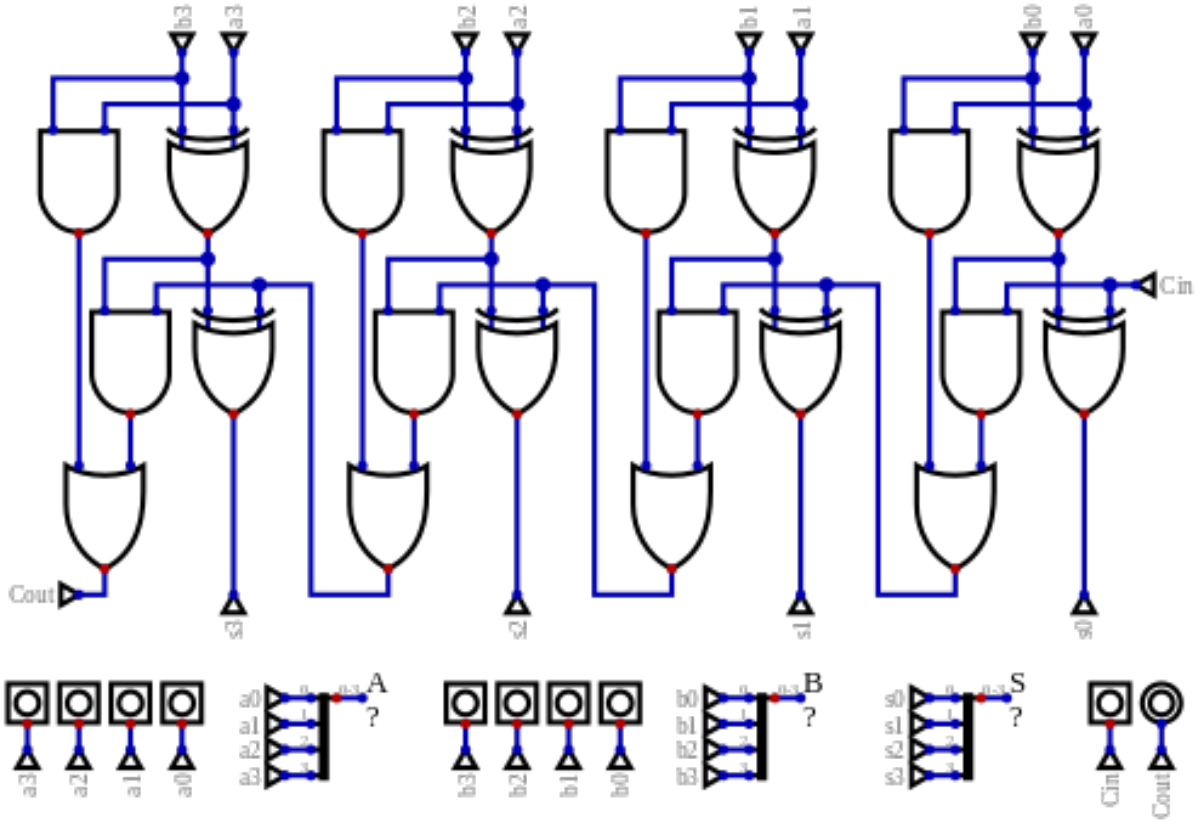


Figura 4.1: Circuito sobre el que visualizar los retardos de las puertas lógicas.

7. Pulsa nuevamente para simular otro paso. Tras este paso ya no hay más puertas afectadas y S ya vale 1. El circuito se ha estabilizado tras 2 pasos (2 niveles de puertas) y el simulador ya no permite más pasos puerta a puerta.

8. ¿Cuántas puertas hay entre la señal $s0$ y las entradas? ¿Y entre la señal $s1$ y las entradas?

9. Con el circuito estabilizado para $A = 0$ y $B = 1$, cambia A a 1 para que cambien los bits $s0$ y $s1$.

10. ¿Por qué $s1$ no se ha actualizado con un retardo igual al número de puertas de su camino con más puertas? (Pista: el retardo observado al estar sumando dos números y a continuación pasar a sumar los dos mismos números será 0, independientemente del número de puertas)

11. Intenta encontrar un caso que genere el retardo máximo. Anota cuál es ese caso y cuál es el retardo máximo.

Dado que cualquier circuito combinacional (puertas/bloques) tarda cierto tiempo en generar las salidas correctas, no funcionará correctamente si sus entradas cambian a mayor velocidad.

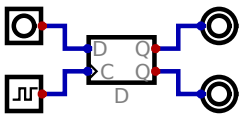
12. Asumiendo que las puertas *and* y *or* tienen un retardo (tiempo de propagación) de 20 ps, y las puertas *xor* de 30 ps, calcula el retardo máximo para cada bit de salida.

    $MaxT(S_0) =$

    $MaxT(S_1) =$

    $MaxT(S_2) =$

    $MaxT(S_3) =$



D	Q	C	Q^+
0	0	↑	
0	1	↑	
1	0	↑	
1	1	↑	

Figura 4.2: Flip-flop D y su tabla de transición. Q indica el estado/salida antes del flanco ascendente de reloj y Q^+ el estado/salida tras el flanco.

$MaxT(Cout) =$

13. Si se desea utilizar el circuito dentro de otro que genere operandos y recoja resultados cada cierto tiempo, ¿cuál será su frecuencia máxima de operación?

4.2. Flip-flops

En general, el estado de un circuito secuencial se almacena en flip-flops, que permiten su cambio de estado en uno de los flancos de reloj. Comprueba el funcionamiento de un flip-flop tipo D:

14. Crea un circuito nuevo, coloca un flip-flop de tipo D (en Flip-Flops) y conecta sus entradas y salidas tal y como se muestra en la figura 4.2.

15. Comprueba su funcionamiento cambiando entradas, completa su tabla de transición y verifica que la tabla equivale a la vista en clase.

4.3. Análisis secuencial

Realiza el análisis secuencial (*en papel*) para el circuito de la figura 4.3:

16. Obtén las *funciones de entrada de los flip-flops* y las *funciones de salida* del circuito:

$J_2 =$
$K_2 =$
$D_1 =$
$D_0 =$
$S_3 =$
$S_2 =$
$S_1 =$
$S_0 =$

17. ¿Es un circuito tipo Moore o tipo Mealy?

18. Obtén las *funciones de transición* (estado futuro en función de entradas y estado actual) de los flip-flops a partir de la ecuación característica de cada flip-flop y las funciones de entrada de los flip-flops:

$Q_2^+ =$
$Q_1^+ =$
$Q_0^+ =$

19. Rellena las tablas de verdad (figura 4.4), especificando estados futuros y salidas *actuales* a partir de estados y entradas actuales.

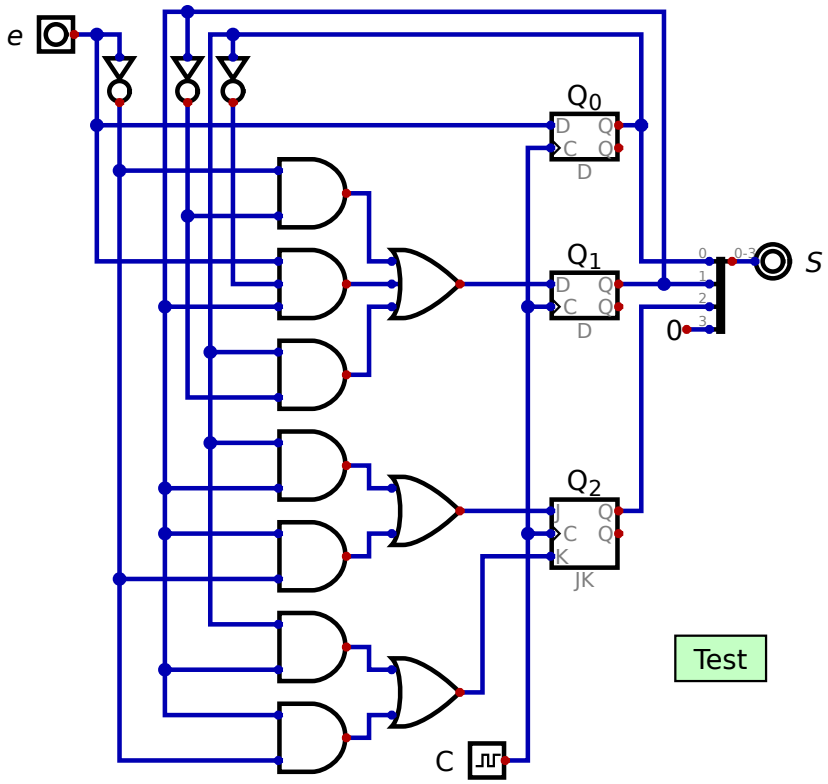


Figura 4.3: Circuito secuencial a analizar.

20. ¿Por qué en este caso es más conveniente separar las tablas de transición y salida en lugar de trabajar con una única tabla?

21. En la figura 4.5, dibuja el autómata o diagrama de estados a partir de las tablas de verdad.

22. Según el diagrama, ¿qué hace el circuito?

23. Dibuja el circuito en *Digital* con nombre `AnalisisSec.dig`. No olvides etiquetar las entradas y salidas de la misma forma.

24. Si asumimos un retardo de 5 ps para las puertas *not* y 20 ps para el resto de puertas, y que los flip-flops tienen un tiempo de setup de 30 ps y un retardo de 50 ps, calcula la frecuencia de reloj máxima (GHz) a la que puede funcionar el circuito.

e	Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

Q_2	Q_1	Q_0	S_3	S_2	S_1	S_0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Figura 4.4: Tablas de verdad (tabla de transición y tabla de salida) del circuito de la figura 4.3.

Figura 4.5: Espacio para dibujar el diagrama de estados del circuito secuencial a analizar.

Práctica 5

Diseño de sistemas secuenciales

En esta práctica se comprobarán las diferencias entre circuitos Moore y Mealy, y se diseñará un circuito secuencial a partir de especificaciones textuales.

5.1. Síntesis tipo Moore y tipo Mealy

Los diagramas Moore y Mealy de la figura 5.1 corresponden a un sistema que pone su salida S a 1 al detectar en la entrada tres 0s consecutivos, sin solapamiento. Sintetiza un circuito secuencial para cada uno de los diagramas.

1. Rellena las tablas de verdad (tabla 5.1) asumiendo flip-flops D
2. Minimiza las funciones de entrada de los flip-flops y las funciones de salida para el diagrama 5.1a:

   $D_1 =$

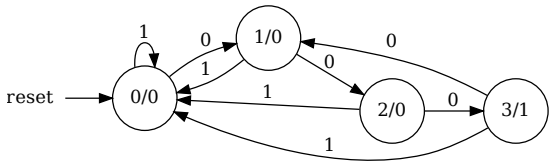
   $D_0 =$

   $S =$
3. Minimiza las funciones de entrada de los flip-flops y las funciones de salida para el diagrama 5.1b:

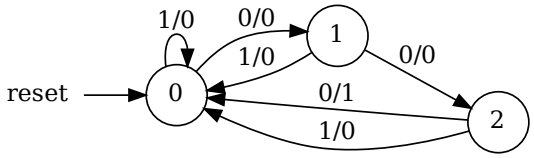
   $D_1 =$

   $D_0 =$

   $S =$
4. Dibuja ambos circuitos por separado con nombres `CerosMoore.dig` y `CerosMealy.dig`. Cada circuito debe tener dos salidas: la propia salida S y su estado Q (2 bits). También deben tener una entrada e y una señal de reloj etiquetada como C.
5. Crea un nuevo circuito (`CerosMooreMealy.dig`) para comparar las versiones Moore y Mealy tal y como se muestra en la figura 5.2. Hay que usar una única señal de reloj a 1 Hz con la marca de "Iniciar el reloj de tiempo real".
6. Inicia la simulación del circuito visualizando el *gráfico de medidas* (cronograma). Ten en cuenta que el cronograma de *Digital* no se muestra respecto al tiempo sino respecto a los cambios de las señales en el tiempo. A efectos prácticos, es como tener un *autozoom* que expande el gráfico cuando hay cambios y lo contrae cuando no los hay. Observa que, si e pasa de 1 a 0, la versión Moore activa la salida S tras 3 ciclos completos con $e = 0$, mientras que la versión Mealy lo hace en tercer ciclo con $e = 0$.
7. Si con $e = 0$ se producen *glitches* (cambios cortos a $e = 1$ fuera de los flancos ascendentes de reloj) cuando $S = 0$, ¿afecta a la salida en algún circuito?



(a) Diagrama Moore



(b) Diagrama Mealy

Figura 5.1: Autómatas o Máquinas de estados finitos para detectar tres 0s consecutivos.

e	Q_1	Q_0	Q_1^+	Q_0^+	D_1	D_0	S
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

(a) Tabla de verdad para el diagrama 5.1a.

e	Q_1	Q_0	Q_1^+	Q_0^+	D_1	D_0	S
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

(b) Tabla de verdad para el diagrama 5.1b.

Tabla 5.1: Tablas de verdad (transición, flip-flops, salida) de los circuitos a sintetizar.

8. Si con $e = 0$ se producen *glitches* cuando $S = 1$, ¿afecta a la salida en algún circuito?

9. ¿Es posible manipular la entrada de forma que el circuito Mealy marque la detección de tres ceros y el circuito Moore no? ¿Cómo?

10. Según el comportamiento observado, rellena el cronograma de la figura 5.3

5.2. Diseño de sistemas secuenciales

Se quiere realizar un circuito receptor de código Morse. Dicho circuito debe monitorizar una señal digital (e) en el tiempo, y debe indicar en sus salidas (P, R, L) si se ha recibido un punto ($P = 1$), una raya ($R = 1$) o se ha completado una letra ($L = 1$). Dependiendo del tiempo que e esté activa (1) tendremos un punto (señal activa sólo durante un ciclo) o una raya (señal activa durante dos o más ciclos), y dependiendo del tiempo que esté inactiva (0) tendremos una separación entre puntos y rayas (señal inactiva sólo durante un ciclo) o una separación entre letras (señal inactiva durante dos o más ciclos).

Es un requisito *imprescindible* el no replicar la información generada. Es decir, si tenemos la señal activa durante 4 ciclos, el circuito debe generar una única salida de raya (hasta el siguiente flanco) y no varias consecutivas (más de un ciclo de duración). Lo mismo es aplicable para la detección de letra completada con varios ciclos con la señal inactiva. Otro requisito es no retrasar innecesariamente la generación de información, es decir, si por ejemplo se ha detectado una raya, hay que generar la salida correspondiente cuanto antes, y no posponerla. Además, la salida debe adaptarse a la entrada en el momento en que ésta se produzca.

Realiza el diseño del circuito descrito con flip-flops JK:

11. ¿Se pide un circuito tipo Moore o tipo Mealy?

12. En la figura 5.4, dibuja el diagrama de estados. (Pista: se necesitan 4 estados)

13. Rellena la tabla 5.2.

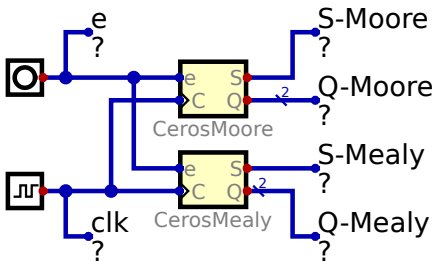


Figura 5.2: Comparador de versiones Moore y Mealy.

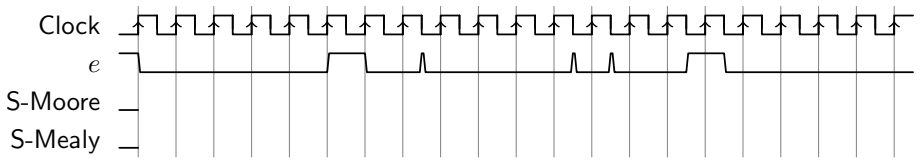


Figura 5.3: Cronograma a rellenar.

14. ¿Por qué en este caso es más conveniente usar una única tabla en lugar de separar las tablas de transición y salida?

15. Minimiza las funciones de entrada de los flip-flops y las funciones de salida (observa que la mitad de los valores serán $\times$ que ayudarán a la minimización).

    $J_1 =$

    $K_1 =$

    $J_0 =$

    $K_0 =$

    $P =$

    $R =$

    $L =$

16. Dibuja el circuito resultante con el nombre `Morse.dig` y verifica su funcionamiento. Recuerda etiquetar entradas y salidas como corresponda, y el reloj como C.

Figura 5.4: Espacio para dibujar el diagrama de estados del circuito secuencial a diseñar.

e	Q_1	Q_0	Q_1^+	Q_0^+	P	R	L	J_1	K_1	J_0	K_0
0	0	0									
0	0	1									
0	1	0									
0	1	1									
1	0	0									
1	0	1									
1	1	0									
1	1	1									

Tabla 5.2: Tabla de verdad (transición, salida, flip-flops) del circuito a diseñar.

Práctica 6

La Máquina Sencilla

En esta práctica vamos a afianzar lo aprendido acerca de la Máquina Sencilla.

6.1. Compilación

La compilación es el proceso en el que un programa expresado en un lenguaje de alto nivel se traduce al lenguaje que entiende el procesador donde se va a ejecutar. En esta práctica, el compilador eres tú.

1. Completa el ensamblador del código 6.2 para que sea equivalente al programa en alto nivel del código 6.1.

2. A partir del ensamblador, completa la tabla 6.1 con la traducción del ensamblador a binario y hexadecimal y su ubicación en memoria. Es *muy importante* tener claro en qué dirección de memoria estarán las instrucciones y los datos.

6.2. Simulación de la Máquina Sencilla

Para probar el programa anterior disponemos de una implementación de la Máquina Sencilla (fichero `MS_cableada_original.dig`) para *Digital* (figura 6.1). Identifica todos los componentes de la unidad de proceso. Observa que hemos diseñado la unidad de control mediante dos ROMs, una para la tabla de transición y otra para la tabla de salidas. Observa también que el bus de datos bidireccional de la RAM obliga a usar puertas triestado controladas por la línea $\overline{L}/E$ para evitar cortocircuitos. Además se ha incorporado un contador de ciclos y un contador de instrucciones (observa qué línea lo activa). También hay visualizadores para observar el valor actual de IR (CO, F y D). Al simular es conveniente visualizar el cronograma (en el menú Simulación→Mostrar gráfico de medidas) y el contenido de la memoria RAM (clic sobre ella). Con todo ello, en cada ciclo puedes saber qué instrucción se está ejecutando, en qué estado está la unidad de control, qué señales de control se están generando, y qué movimiento de información (acción RTL) se producirá en el cambio de ciclo.

3. Escribe la columna «Hexadecimal» de la tabla 6.1 en un fichero de texto, con un único número hexadecimal (sin el prefijo habitual 0x) en cada línea. Para bits *don't care* deberás especificar un valor concreto, por ejemplo 0. Para repetir un mismo valor n veces puedes escribir $n*valor$. Por ejemplo para escribir 8 posiciones de memoria consecutivas con ceros se puede escribir una única vez `8*0`. La primera línea del fichero debe contener `v2.0 raw`, y se recomienda guardarlo con nombre `NIP_P6a.hex`[1].

[1] Cabe la posibilidad de que la aplicación utilizada guarde el fichero con el nombre «NIP_P6a.hex.txt». Si es así, verás que el contenido de la RAM no se ajusta en absoluto al contenido del fichero. Para solventar el problema, al guardar el fichero hay que especificar como tipo «todos los archivos» para que «.txt» no se añada por defecto al nombre especificado. Siguiendo esta pauta no debería haber discrepancia entre el contenido de la RAM y el contenido del fichero.

```
// Declaración de variables
int a=3;          // primer operando, inicializado con valor 3
int b=4;          // segundo operando, inicializado con valor 4
int c;            // resultado del cálculo c=a·b=∑_{i=0}^{b-1} a, sin inicializar
int i;            // variable de inducción del bucle, sin inicializar

// Código ejecutable
c=0;              // asignación del valor 0
i=0;              // asignación del valor 0
while (i!=b){     // condición: iterar mientras no hayamos hecho b iteraciones
  c=c+a;          // incrementar c con a
  i=i+1;          // incrementar variable de inducción
}
halt();
```

Código 6.1: Programa en alto nivel.

```
.data 16                  ; datos situados a partir de la dirección 16 (decimal)
a:      .dw 3             ; .dw reserva una palabra (16 b) con cierto valor
b:      .dw 4             ;
c:      .rw 1             ; .rw 1 reserva una palabra sin inicializar
i:      .rw 1
cero:   .dw 0
uno:    .dw 1

.code                     ; inicio de código (a partir de la dirección 0)
        mov cero, c       ; c=0
        mov cero, i       ; i=0
WHILE:                    ; si (i==b)
                          ;   salta a HALT
                          ; c=c+a;
                          ; i=i+1;
                          ; comparación para forzar
                          ;   salto a WHILE (incondicional)
HALT:   cmp cero, cero    ; comparación para forzar
        beq HALT          ;   salto a HALT (bucle infinito)
.end
```

Código 6.2: Programa en ensamblador de la Máquina Sencilla.

	CO	dir. F	dir. D	Binario (16 b)	Hexadecimal
@0	2	20	18	10 0010100 0010010	8A12
@1					
@2					
@3					
@4					
@5					
@6					
@7					
@8					
@9					

	Hex
@16	
@17	
@18	
@19	
@20	
@21	

Tabla 6.1: Contenido del programa (instrucciones y datos) en la memoria de la Máquina Sencilla.

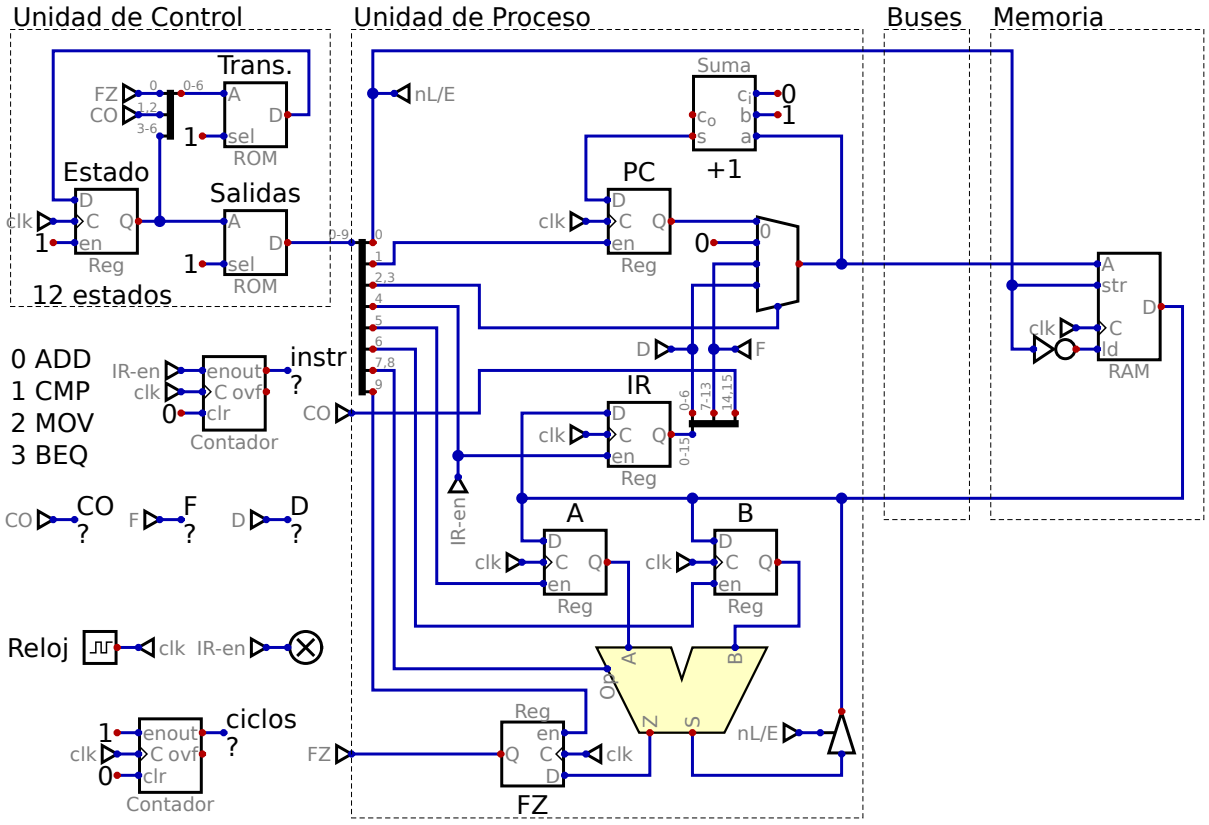


Figura 6.1: Diseño de la Máquina Sencilla

4. Para que la RAM se precargue cada vez que se inicia la simulación, hay que especificarlo desde el menú Editar→Ajustes específicos del circuito→Avanzado→Precarga la memoria del programa al comienzo.

5. Inicia la simulación de la Máquina Sencilla. Dispones de tres opciones distintas de simulación:

   a) Simulación ciclo a ciclo, pulsando sobre la señal de reloj.

   b) Simulación instrucción a instrucción, pulsando ▶▶|.

   c) Simulación continua, sin paradas, si editas el reloj y marcas «Iniciar el reloj de tiempo real». En este último caso deberás analizar el cronograma tras parar la simulación.

6. Simula tu programa hasta llegar a *HALT* por primera vez (la variable `c` debería contener $3 \times 4 = 12$) y rellena la tabla 6.2. Dado que la instrucción BEQ puede ahorrar un ciclo a la instrucción siguiente, para que cuadren todos los números vamos a considerar que el ciclo ahorrado corresponde a la propia instrucción BEQ.

7. Rellena de la misma forma la tabla 6.3, pero ahora usando la Máquina Sencilla *optimizada* (fichero `MS_cableada_optimizada.dig`).

8. Si el reloj del procesador fuera a 16 Hz, ¿cuánto tiempo costaría ejecutar el programa en estas dos versiones de la Máquina Sencilla?

6.3. Un programa más complejo

Vamos a repetir lo anterior, pero ahora con un programa que necesita acceder a un vector a través de un índice (direccionamiento relativo). Dado que la Máquina Sencilla no dispone de un modo de

Instrucción	Veces ejecutada	Ciclos por instrucción	Total (veces × ciclos)
ADD CMP MOV BEQ			
Total ciclos del programa:			

Tabla 6.2: Resumen de la ejecución del código 6.2 en la Máquina Sencilla original.

Instrucción	Veces ejecutada	Ciclos por instrucción	Total (veces × ciclos)
ADD CMP MOV BEQ			
Total ciclos del programa:			

Tabla 6.3: Resumen de la ejecución del código 6.2 en la Máquina Sencilla optimizada.

direccionamiento apropiado para recorrer vectores, en cada iteración del bucle es necesario incrementar el campo F de la instrucción que lee el elemento del vector. La instrucción `add cons, INST` de la figura 6.4 modifica la instrucción que accede al vector, sumando 1 al campo F de la instrucción situada en la posición INST. Esta técnica (*código automodificable*) se utilizaba en los primeros computadores (años 1940–60), precisamente por las mismas limitaciones que observamos en la Máquina Sencilla. En la actualidad sigue usándose para ciertos propósitos, como por ejemplo enlazar código de bibliotecas dinámicas o dificultar (*ofuscar*) la comprensión del código programado.

9. Completa el ensamblador del código 6.4 para que sea equivalente al programa en alto nivel del código 6.3.

10. Completa la tabla 6.4 con la traducción del ensamblador a binario y hexadecimal.

11. Simula el programa ciclo a ciclo con la Máquina Sencilla *original*, prestando especial atención a la ejecución de la instrucción que modifica la instrucción INST.

12. Comprueba que el resultado (dato final en la variable suma) es correcto.

13. Rellena la tabla 6.5. Igual que antes, hay que asumir que el programa acaba cuando llega a *HALT* por primera vez.

```
int vector={1,7,12,56,-12,-3,-10,0}; // vector de números a acumular
int suma=0; // variable donde quedará el resultado
int i;      // índice de acceso al vector
int num;    // variable donde obtenemos el número a sumar

i=0;
num=vector[i];
while (num!=0){ // mientras num≠0
  suma=suma+num;
  i=i+1;
  num=vector[i];
}
halt();
```

Código 6.3: Programa «complejo» en alto nivel.

```
.data   32                ; datos situados a partir de la dirección 32 (decimal)
vector: .dw 1,7,12,56,-12,-3,-10,0 ; valores de 16 b consecutivos en memoria
suma:   .dw 0
num:    .dw 1             ; la variable i no se utiliza
cero:   .dw 0
uno:    .dw 1
cons:   .dw 128

.code                     ; inicio de código (a partir de la dirección 0)
        mov vector, num   ; num=vector[0]
WHILE:                    ; si num==0
                          ;   salta a HALT
                          ; suma=suma+num;
        add cons, INST    ; modificar instrucción de acceso a vector
INST:   mov vector, num   ; num=vector[i]
                          ; comparación para forzar
                          ;   salto a WHILE (incondicional)
HALT:                     ; comparación para forzar
                          ;   salto a HALT (bucle infinito)
.end
```

Código 6.4: Programa «complejo» en ensamblador de la Máquina Sencilla.

	CO	dir. F	dir. D	Binario (16 b)	Hexadecimal
@0	2	32	41	10 0100000 0101001	9029
@1					
@2					
@3					
@4					
@5					
@6					
@7					
@8					
@9					

	Hex
@32	
@33	
@34	
@35	
@36	
@37	
@38	
@39	
@40	
@41	
@42	
@43	
@44	

Tabla 6.4: Contenido del programa «complejo» en la memoria de la Máquina Sencilla.

Instrucción	Veces ejecutada	Ciclos por instrucción	Total (veces × ciclos)
ADD CMP MOV BEQ			
Total ciclos del programa:			

Tabla 6.5: Resumen de la ejecución del código 6.4 en la Máquina Sencilla original.

Trabajo práctico: la Máquina Sencilla con pantalla

Para realizar el trabajo práctico es imprescindible entender el funcionamiento de la Máquina Sencilla. Por ello, es muy importante completar la práctica 6 antes de empezar el trabajo práctico.

Para el trabajo se ha modificado la Máquina Sencilla añadiendo una pantalla de 8×8 píxeles y su correspondiente controlador. Para iluminar un píxel hay que escribir en la dirección de memoria 127 el código (16 bits) del color a usar. Los píxeles se iluminan en secuencia, por filas, de izquierda a derecha. Como ejemplo, en Moodle dispones de tres imágenes 8×8 ya codificadas como una secuencia de colores. Puedes usar cualquiera de ellas, o crear otra. La especificación de los colores está detallada en el circuito controlador.

RAM para la Máquina Sencilla original

Realiza un programa en ensamblador para la Máquina Sencilla (fichero `MS_trabajo1.dig`) que dibuje una imagen en la pantalla. El programa debe empezar en la dirección de memoria *NIP* mód 35, y los datos (incluyendo la imagen a dibujar) deben empezar en la dirección de memoria (*NIP* mód 35) + 15. Por ejemplo, para el NIP 123456 la primera instrucción del programa debe estar en la dirección 123456 mód 35 = 11 y la primera variable en la dirección 26.

RAM para la Máquina Sencilla microprogramada

Además de la modificación anterior, se ha modificado el microprograma de la Máquina Sencilla microprogramada y se ha añadido una conexión desde la ALU al multiplexor de direcciones (figura 2 y fichero `MS_trabajo2.dig` en Moodle).

1. Interpreta el nuevo microprograma para saber qué hacen ahora las instrucciones
2. Realiza un programa equivalente al del apartado anterior *sin usar código automodificable*. Igual que antes, el programa debe empezar en la dirección de memoria *NIP* mód 35 y los datos en (*NIP* mód 35) + 15.

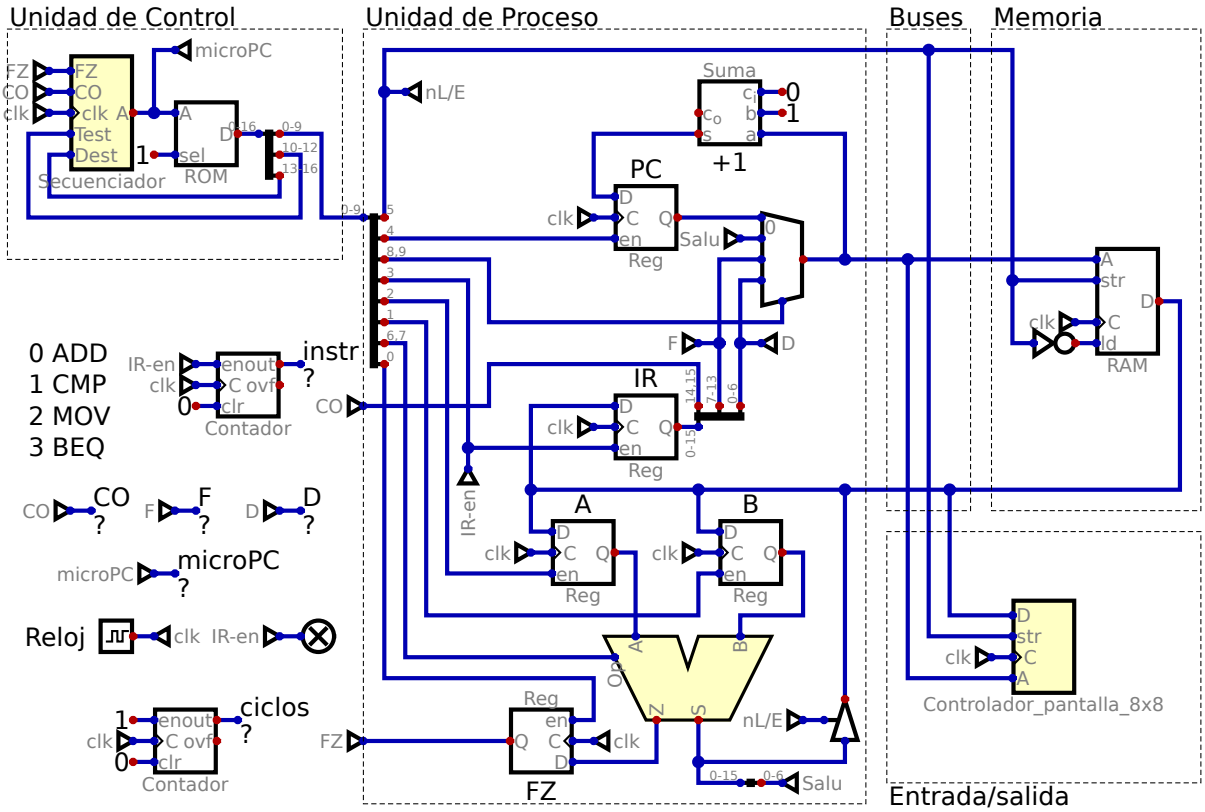


Figura 2: Máquina Sencilla microprogramada modificada.