Material de apoyo para la resolución de ejercicios de Diseño Lógico

Instituto de Ingeniería Eléctrica - Grupo de Diseño Lógico
Facultad de Ingeniería
Universidad de la República

7 de julio de 2008
Montevideo, Uruguay

Índice general

Capítulo 1

Introducción

1.1. Presentación de este material

Consideramos que un buen apoyo para terminar de comprender la globalidad de la asignatura y para preparar las pruebas finales, consiste en resolver problemas de diseño completos de características similares a los de los exámenes. Actualmente están disponibles todos los ejercicios resueltos planteados en exámenes de años anteriores.

Para complementar y potenciar ese material existente, decidimos seleccionar algunos ejercicios cuya resolución plantea problemas interesantes, y comentar la misma.

1.2. Organización y detalles

Para esta primer versión del material seleccionamos 3 problemas de RTL de distintos años. Cada problema comienza con el planteo de la letra, y luego se presenta la solución comentada del mismo. Se agregaron comentarios, observaciones, explicaciones de errores típicos y diagramas de tiempos, con la intención de mejorar la comprensión de los diseños realizados.

Además se dejan disponibles en la página del curso los archivos .gdf de todos los circuitos diseñados, así como también los archivos .scf con las simulaciones. De esta manera puede fácilmente simularse el funcionamiento del circuito con otras entradas, observar otras señales que puedan ser de interés, o ver si circuitos distintos se comportan de la misma manera. Dado que un mismo problema admite generalmente distintas soluciones, este último punto facilita la auto evaluación de diseños distintos a los presentados.

Tenemos previsto completar este material con ejercicios de los otros temas del curso, toda sugerencia o comentario para mejorar el presente material es bienvenido.

Capítulo 2

Lenguaje de Diseño RTL

2.1. Examen de Marzo de 2002 - Problema 1

2.1.1. Introducción

En este problema se ve un ejemplo de cómo ir decodificando un código serial, formando una salida paralela. En este caso se trata de un tipo de codificación que se usa en el protocolo USB. La idea del relleno de bits que se usa en la misma es muy común en el ámbito de los protocolos de comunicación.

2.1.2. Letra del problema

La comunicación USB (Universal Serial Bus) emplea la codificación de datos NRZI (Non Return to Zero Invert) con bit de relleno (bit stuffing) a los efectos de poder transmitir datos y sincronismo (reloj) por una única línea.

Esta codificación consiste en lo siguiente:

. Un "1" se representa manteniendo el nivel del bit anterior.

. Un "0" se representa invirtiendo el nivel del bit anterior.

. Para asegurar al menos 1 transición en la señal cuando hay varios "1's" consecutivos, el transmisor inserta un "0" de relleno (o sea una transición forzada) luego de cada 6 "1's" consecutivos. Si son 6 "0s" consecutivos es lo mismo (se rellena con un 1).

Esta codificación le da a la lógica del receptor una transición en la línea de datos al menos cada 7 tiempos de bit, lo que garantiza que un bloque "regenerador de reloj" pueda reconstruir el reloj del transmisor que será utilizado por el receptor (ver figura).

Veamos el siguiente ejemplo para enviar el dato hexadecimal: 6 7 E 7 F F E 1

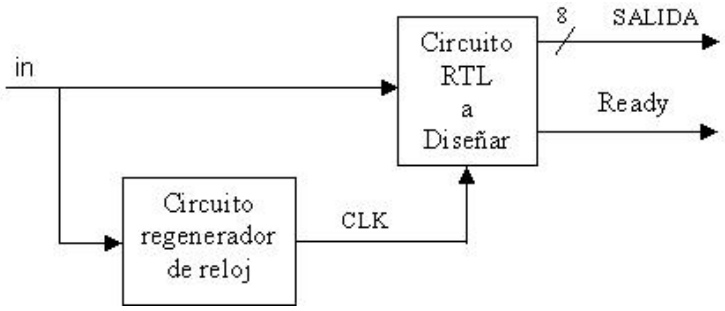
El receptor debe decodificar la señal, reconocer el "bit de relleno" y descartarlo.

Diseñe un circuito RTL receptor de USB que decodifique la entrada **in** y tenga 2 salidas que se comportan como se explica a continuación:

```
Ej:                 6    7    E     7    F     F    E     1
Dato a transmitir:  0110 0111 111 0 0111 111 1 1111 1 110 0001
Señal enviada:      1110 1111 11101 0000 00011 1111 10001 0100
                                    Bit de relleno
```

. SALIDA[8]: Salida paralelo de los bits de entrada decodificados. Considere que el transmisor envía primero el MSb en cada byte transmitido. Debe ser válida sólo cuando Ready = 1.

. Ready: Debe subir a 1 durante un período de reloj para indicar que hay un dato válido en SALIDA[].



Suponga al inicio que el "nivel anterior" fue un "0" y que a partir de este momento el transmisor continuamente envía datos.

Se pide la secuencia RTL y el circuito completo.

2.1.3. Solución

Algunas consideraciones

- Observar que el bit stuffing se realiza en el dato a transmitir, es decir, en la señal enviada (la entrada in de nuestro circuito) pueden haber hasta 7 "1s" o "0s" consecutivos.
- Para realizar la decodificación, observamos lo siguiente: si nos llegan dos bits consecutivos con el mismo valor, quiere decir que se mantuvo el valor, y esto tiene que ser consecuencia de un 1 en la entrada, por otro lado, si el valor cambió, esto significa que la entrada era un 0. Por lo tanto, lo que necesitamos es poder determinar si dos bits consecutivos son iguales, independientemente del valor de estos bits.

Memoria necesaria para la implementación

- Necesitamos un registro para ir formando el byte de salida (MEM[7..0]) y un contador de bits que llegue hasta 8 para saber cuándo terminó de llegar en forma serial cada byte (CONT8[2..0]).

- Para eliminar los bits de relleno, necesitamos un contador de "1s" o "0s" que cuente hasta 6 repeticiones (CONT6[2..0]). Ese último contador tiene que ser independiente del primero porque los bits iguales consecutivos pueden estar en distintos bytes. CONT6[2..0] cuenta las repeticiones, por lo que cuando llega a 6 significa que hubo 7 bits iguales consecutivos.

- Para determinar si dos bits consecutivos son iguales, definimos un registro que guarde el bit anterior (`IN_OLD`) de forma de poder ir comparando el bit de entrada con el bit anterior. Sin embargo, también es válido realizar una bifurcación e ir a pasos distintos dependiendo del valor del bit anterior (se utilizaría un registro más en el bloque de control, pero no se necesitaría el registro que usamos para almacenar el valor anterior); en esta solución presentamos la primer opción.

- MEM va a ser un Shift Register (ver sección de errores típicos 2.1.4) para transformar de serie a paralelo (notar que primero se recibe el MSB), pero en vez de poner el dato recibido se debe poner "1" si el dato anterior y el actual son iguales (`IN_OLD` = in) y "0" si son distintos (`IN_OLD`≠in). Podemos definir una función lógica que implemente esto: $iguales = !(InOld \oplus in)$.

Inicialización

- Se supone que "el nivel anterior" al inicio fue cero, entonces en el estado inicial en lugar de utilizar `IN_OLD`, tomamos 0 directamente como entrada anterior.

- El CONTADOR8 no necesita reinicializarse gracias al "INC" cíclico, pero el CONTADOR6 sí lo necesita porque no llega al máximo que le permiten sus 3 bits ($2^3 - 1 = 7$).

Secuencia RTL

```
MODULE:  EXAMEN
INPUTS: IN
OUTPUTS: SAL[8], ready
MEMORY: MEM[8], IN_OLD, CONT6[3], CONT8[3].
BUSES: iguales

1  MEM[7..0] <- 0000000,NOT(IN)            ;Se supuso que la entrada anterior
                                           ;era 0, entonces "iguales" pasa a
                                           ;ser not(IN).
   CONT8 <- 001                            ;Se recibio un bit.
   CONT6 <- 001 and NOT(IN)                ;Si el "IN" actual es igual al
                                           ;anterior (que se supone 0
                                           ;inicialmente), incremento el
                                           ;contador de "iguales consecutivos"

2  MEM[7..0] <- MEM[6..0],iguales          ;Transfiero los bits "traducidos".
   CONT8 <- INC(CONT8)                     ;Se recibio un bit.
   CONT6 <- INC(CONT6) and iguales         ;Esto hace que CONT6 se incremente
                                           ;siempre que los bits sigan siendo
                                           ;iguales. Dos bits consecutivos
                                           ;distintos lo llevan inmediatamente
                                           ;a cero.
   -> ((CONT6=5) and iguales), en otro caso) / (3 , 2)

3  CONT6 <- 001 and iguales                ;esto es equivalente a CONT6 <- 0 ya
                                           ;que por construcción del código
                                           ;nunca van a poder venir 8
                                           ;caracteres iguales seguidos.
   -> 2                                    ;Desecho el bit de relleno.

   END SEQUENCE
   CONTROL RESET(1)

   IN_OLD <- IN
   iguales = NOT (IN xor IN_OLD)
   ready = (CONT8 = 0)
   SAL = MEM

   END
```

NOTA: El paso 1 es igual al paso 2 suponiendo `IN_OLD=0`.

Bloque de Control

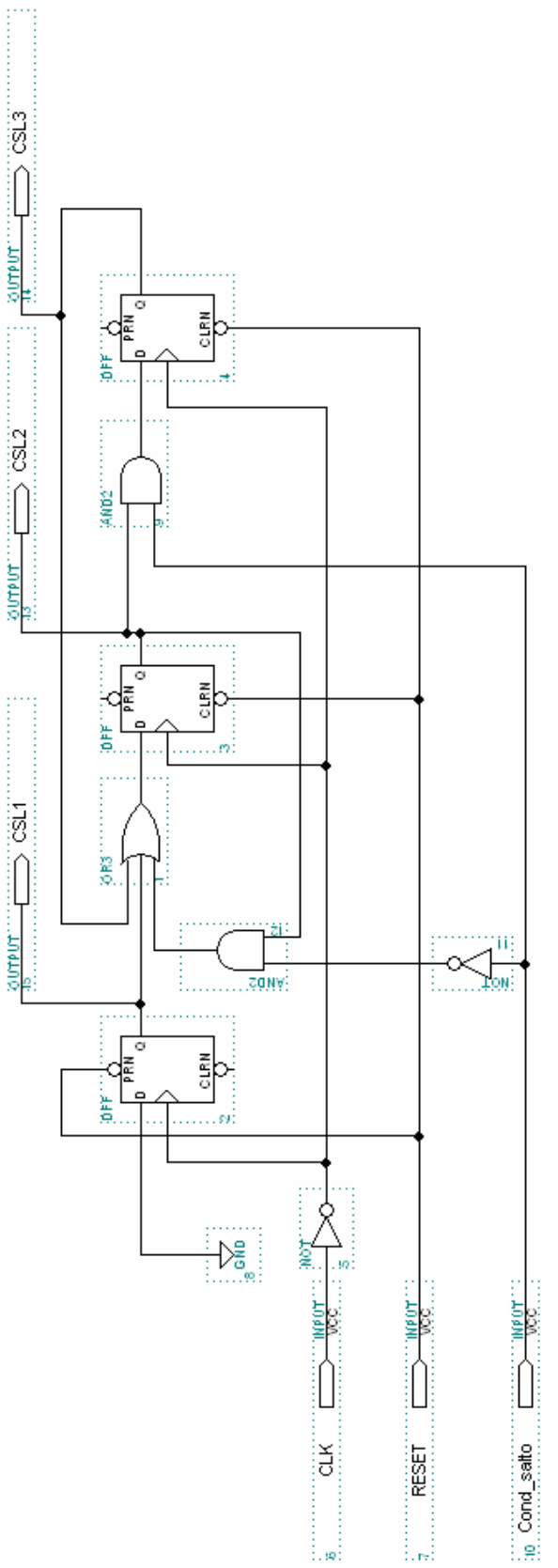


Figura 2.1: Bloque de Control

Bloque de Datos

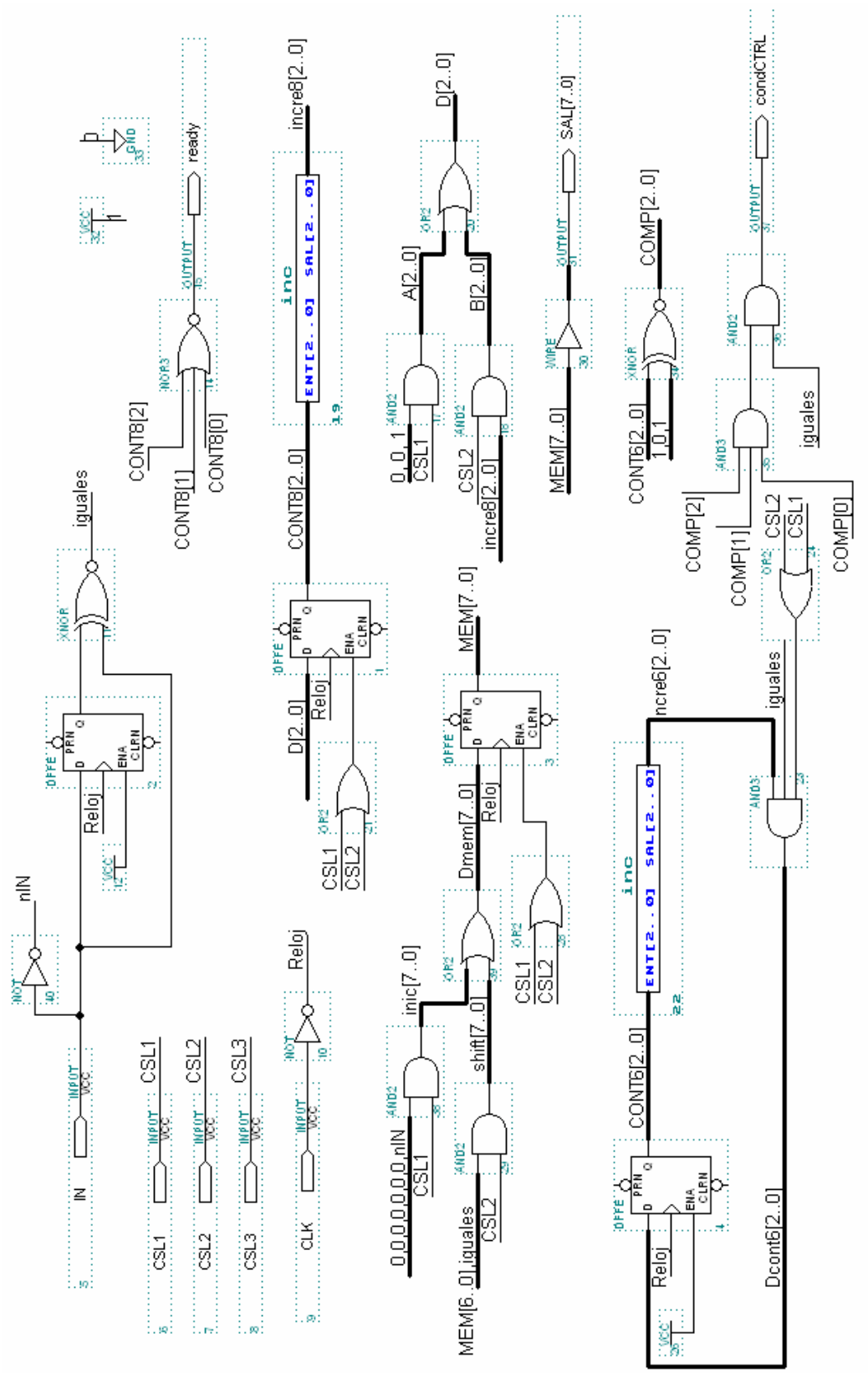


Figura 2.2: Bloque de Datos

Diagrama de tiempos

A continuación en la figura 2.3, se muestra una simulación del circuito RTL diseñado, tomando como entrada la señal enviada del ejemplo. En la figura se pueden ver los primeros 4 dígitos hexadecimales decodificados correctamente, en el archivo de simulación que se adjunta con el presente material se puede ver el resto de la secuencia.

2.1.4. Errores típicos

En este problema es necesario procesar y pasar datos recibidos en forma serial a un formato paralelo (MSB). Tal como se explicó, para la solución se decidió utilizar MEM[7..0] como shif register. Un error típico encontrado en los exámenes es el siguiente:

```
1  i <- 7;
....
2  MEM[i] <- iguales; i <- DEC(i); Transfiero los bits "traducidos".
....
```

El RTL no admite el direccionamiento a través de un índice.

Observar que todas las sentencias RTL tienen un hardware asociado, pero no es inmediato encontrar un hardware asociado en este caso.

En general, un circuito con direccionamiento indexado dentro de un registro podría implementarse utilizando un multiplexor.

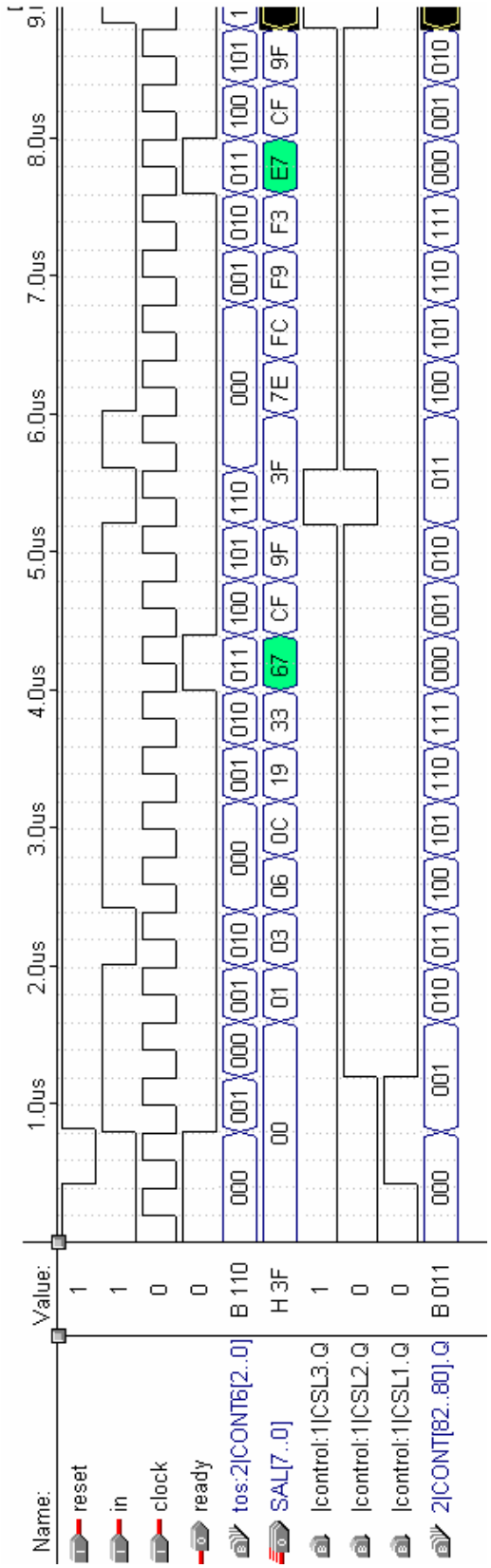


Figura 2.3: Diagrama de tiempos

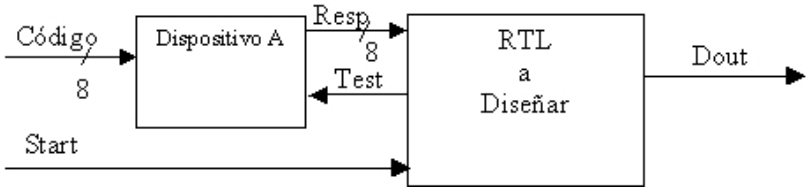
2.2. Examen de Julio de 2003 - Problema 2

2.2.1. Introducción

En este problema se presta especial atención a los aspectos de temporización de las señales y cómo realizar adecuadamente transferencias de funciones lógicas. Se toma como ejemplo el pasaje de un código ternario con largo de palabra 1, a un código binario con largo de palabra 2.

2.2.2. Letra del problema

Se desea diseñar un sistema que lea un Código de un dispositivo externo y lo transmita por una salida serie Dout. El Código está formado por 8 líneas, donde cada una de ellas puede tener 3 valores posibles: 0, 1, TE (tercer estado).



Existe un dispositivo A que convierte de ese código ternario a binario. El dispositivo da salidas por sus patas Resp[i] de acuerdo al valor de Test y Código[i] (donde i es el número de línea) como se ve en la tabla. Hay que utilizar adecuadamente la señal Test para distinguir entre los tres valores de Código.

Código[i]	Test	Resp[i]
0	X	0
1	X	1
TE	0	0
TE	1	1

Para comenzar la transmisión debe esperarse que la entrada sincronizada Start valga uno. En ese instante debe indicarse Dout = 0 (bit de arranque) y en el siguiente período de reloj se debe iniciar la transmisión comenzando por la línea de valor más significativa.

La forma de trasmitir el código por la salida Dout es codificando cada línea de entrada con 2 bits (de un período de reloj cada bit) según la siguiente tabla:

1	11	Dos 1's consecutivos
0	00	Dos 0's consecutivos
TE	10	Un 1 seguido de un 0

De esta forma cada transmisión debe demorar 16 períodos de reloj (sin contar el bit de arranque). Cuando no hay una transmisión en curso, Dout = 1.

Durante la transmisión se ignorará el valor de start.

Tener en cuenta que las trasmisiones se pueden realizar en forma consecutiva de forma tal que inmediatamente después del último bit de la transmisión anterior se deba dar el bit de arranque de la nueva transmisión.

El valor de Código se mantiene estable desde que sube Start a 1 hasta el final de la transmisión por lo que debe ser leído durante este intervalo.

Diseñar un sistema RTL que maneje el dispositivo A para leer Código y que disponga de las entradas y salidas indicadas en la figura. Se pide descripción RTL, bloque de control y de datos.

2.2.3. Solución

Los valores del Código están dados en un sistema ternario. El problema propone transmitir por una salida binaria esta información a partir de una codificación dada. Se debe utilizar el dispositivo A para determinar el valor de cada línea del Código (1, 0 o TE).

La forma de utilizar el Dispositivo A es:

I Se pone a 1 la señal Test y se lee el valor de Resp correspondiente

II Se pone a 0 el valor de Test y se vuelve a leer el valor de Resp. A partir del valor de cada línea de Codigo[] se concluye:

- a Si se leyó 0 las 2 veces en la línea i, entonces el valor de Codigo[i] = 0
- b Si se leyó 1 las 2 veces en la línea i, entonces el valor de Codigo[i] = 1
- c Si se leyó 1 la primera vez y 0 la segunda en la línea i, Codigo[i] = TE
- d Nunca va a pasar que se lea 0 la primera vez y 1 la segunda por definición del Dispositivo A.

De igual forma se podría realizar II) primero y I) en segunda instancia, lo que hará que c) sea con un 0 en la primera lectura y un 1 en la segundo.

La solución propone comenzar por I) y luego II). Esto es debido a la forma de codificar TE (10) ya que de esta forma que da más cómodo para la solución propuesta.

Por otro lado se indica que la transmisión serie debe comenzar con un bit de arranque sincronizado con la señal Start. Esto implica que la señal Dout debe bajar a 0 conjuntamente con la subida de Start.

A continuación se detalla un diagrama de tiempos para aclarar como deben sincronizarse Dout con Start.

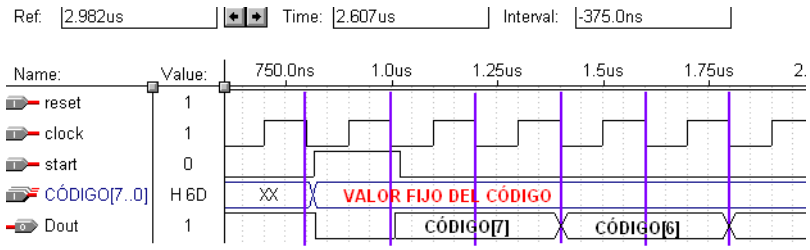


Figura 2.4: Detalle de sincronización

Secuencia RTL

```
MODULE: EXAMEN
INPUTS: Resp[8]
OUTPUTS: Dout, Test
MEMORY: Test_uno[8], Test_cero[8], Cont[3]

1. Dout = !Start                    ; Dout baja cuando sube Start
   Test = 1                         ; Se comienza con Test = 1
   Test_uno <- Resp                 ; Se guarda el resultado del test al final
                                    ; del paso.
   Cont <- 0                        ; Se inicializa el contador
   -> (Start, !Start)/(2,1)

2. Test = 0                         ; Ahora se testea con Test = 0
   Dout = Test_uno[7]               ; bit1 comenzando por la línea más
                                    ; significativa
   Test_cero <- Resp.(Cont = 0) + (Test_cero[6..0],0) . (Cont <> 0)

3. Test_uno <- (Test_uno[6..0],0)
   Cont <- Inc(Cont)
   Dout = Test_cero[7]              ; bit0 comenzando por la línea más
                                    ; significativa
   ->(Cont=111,Cont<>111)/(1,2)

  END SEQUENCE
  CONTROLRESET(1)
  END
```

Bloque de Control

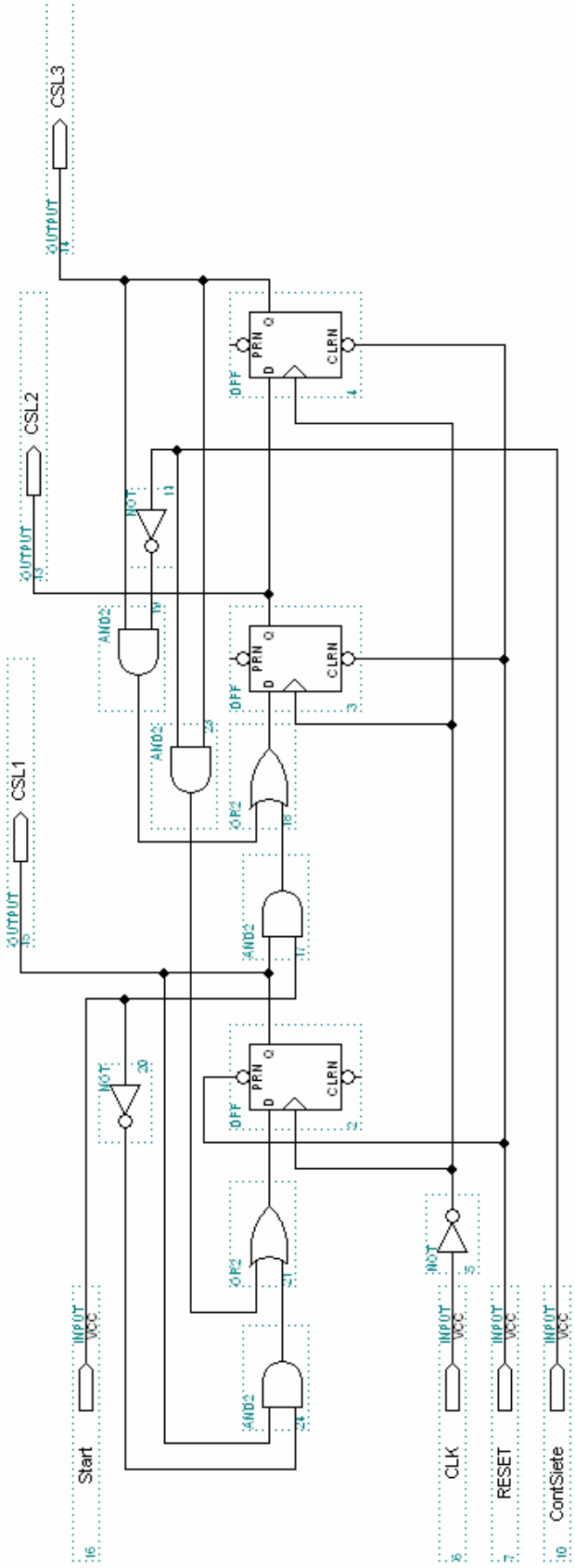


Figura 2.5: Bloque de Control

Bloque de Datos

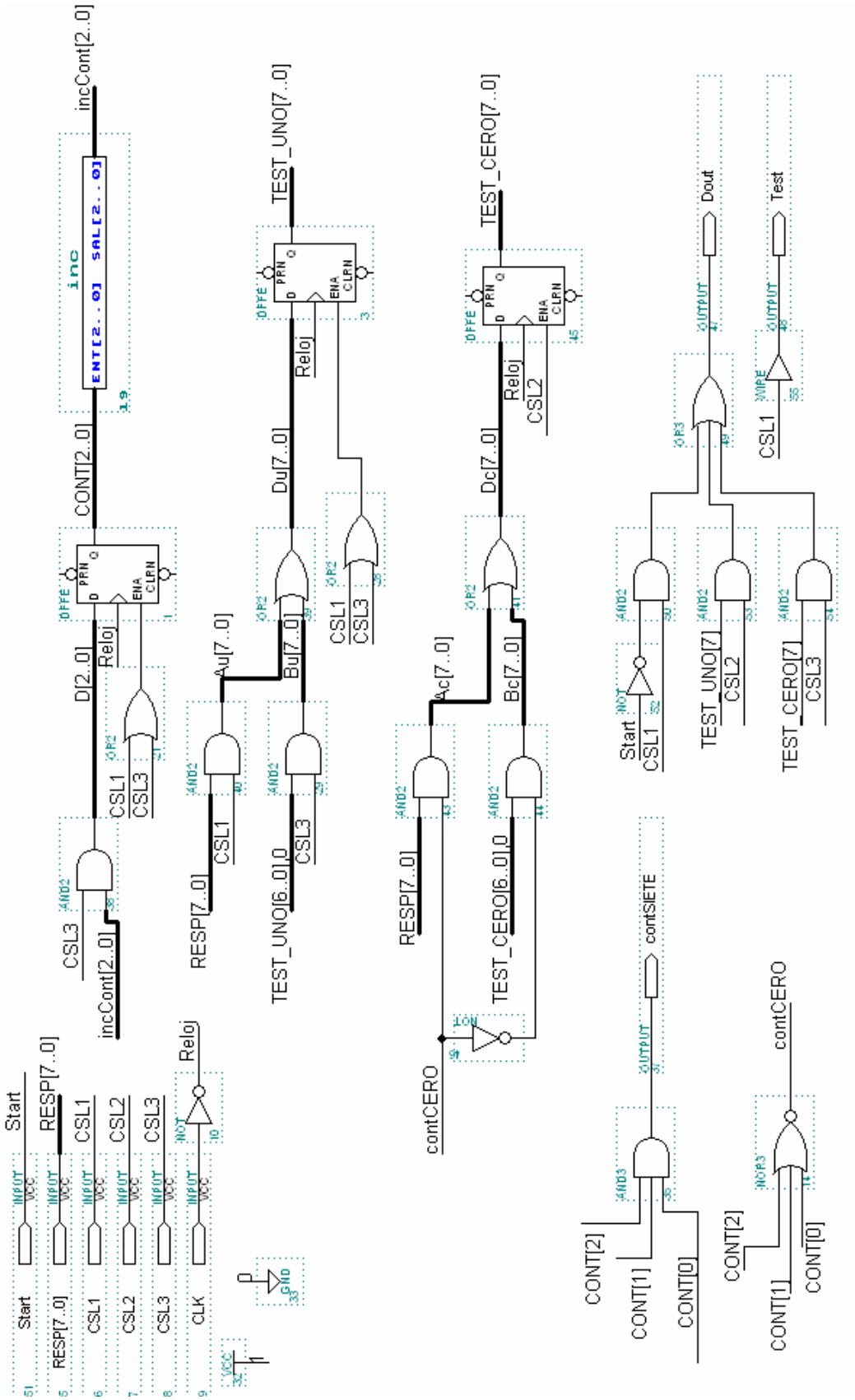


Figura 2.6: Bloque de Datos

Comentarios

Se observa que, a diferencia de Dout, durante el paso 1, Test = 1 siempre. La que diferencia es que cuando la señal Test importa, el valor requerido siempre es 1, por lo que para simplificar se toma de esta forma.

También se observa en el paso 1 que se realizarán las transferencias a `Test_uno` y a Cont en forma incondicional. Esto es correcto, ya que el contenido de estos registros solo importa al dejar el paso 1 (o bien pasar al paso 2). Hacerlo condicional con Start = 1 no estaría mal, pero agregaría lógica en forma innecesaria.

En el paso 2, ocurre lo mismo con Test = 0, solo importa su valor cuando cont = 0

También en el paso 2 se tiene una transferencia:

`Test_cero` ← Resp∧ (Cont = 0) ∨ (`Test_cero[6..1]`,0) ∧ (Cont <> 0)

Con esta sentencia se está transfiriendo Resp la primera vez (en que Cont = 0) y las veces subsiguientes se realiza un corrimiento hacia la izquierda (en que Cont<>0). Observar que no se trata de una transferencia condicional pues la transferencia se realiza siempre, lo que varía es lo que se transfiere.

Luego en el paso 3 solo se realiza el corrimiento a la izquierda pues la transferencia con el contenido de Resp ya se realizó en el paso 1.

Notar que durante la transmisión, en los pasos 2 y 3, se ignora el valor de Start, tal como lo pide la letra del problema.

Diagrama de tiempos

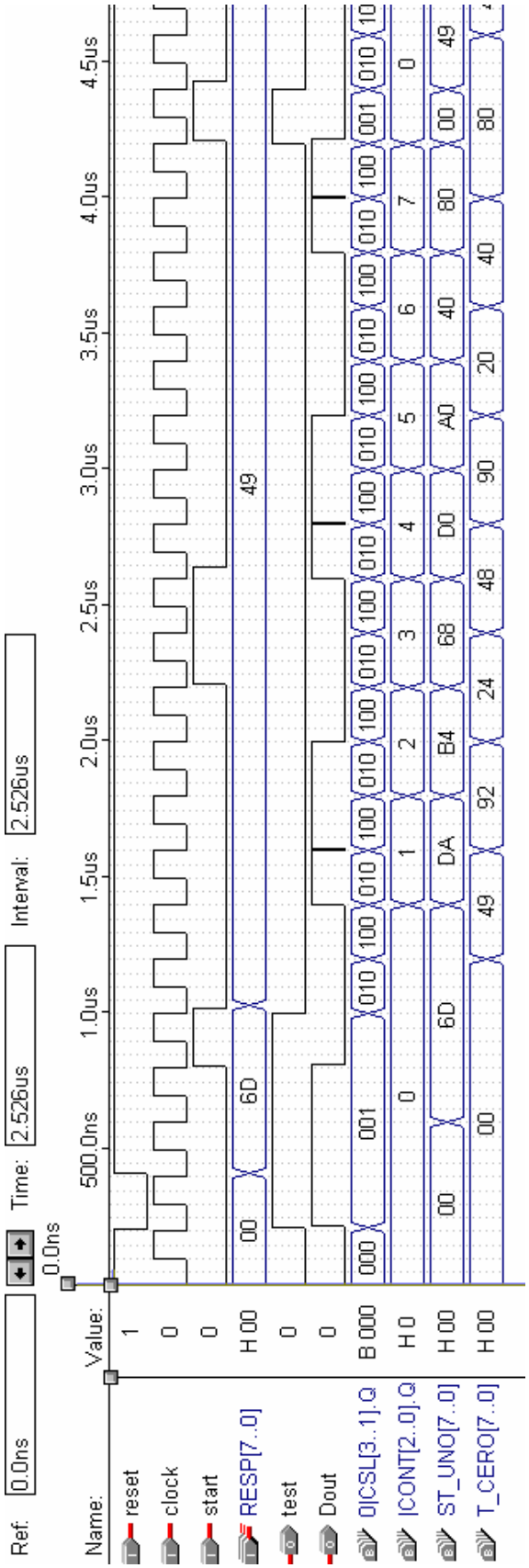


Figura 2.7: Simulación

Otra posible solución podría ser la siguiente:

```
MODULE: EXAMEN
INPUTS: Resp[8]
OUTPUTS: Dout, Test
MEMORY: Test_uno[8], Test_cero[8], Cont[3]

1. Dout = !Start           ; Dout baja cuando sube Start
Test = 1                   ; Se comienza con Test = 1
Test_uno <- Resp            ; Se guarda el resultado del test al final del paso.
Cont <- 0                   ; Se inicializa el contador
-> (Start, !Start)/(2,1)

2. Test = 0                 ; Ahora se testea con Test = 0
Dout = Test_uno[7]          ; bit1 comenzando por la línea más significativa
Test_cero <- Resp
->(4)

3. Dout = Test_uno[7
Test_cero <- Test_cero[6..0],0

4. Test_uno <- (Test_uno[6..0],0)
Cont <- Inc(Cont)
Dout = Test_cero[7]         ; bit0 comenzando por la línea más significativa
-> (Cont=111,Cont<>111)/(1,3)

END SEQUENCE
CONTROLRESET(1)
END
```

En esta solución se separa el paso 2 en dos pasos, uno para obtener el valor de Resp y el otro para el corrimiento. De esta forma no es necesario realizar la transferencia con combinatoria. Se deja como ejercicio la realización del bloque de control y datos para esta solución

2.2.4. Errores típicos

Un error muy común es escribir la transferencia de `Test_Cero` en el paso 2 de la siguiente forma:

2. . . .
`Test_cero * Cont=0` ← Resp
`Test_cero * Cont <> 0` ← `Test_cero[6..0],0`

A primera impresión parecería que la transferencia de arriba se realiza si Cont=0 y la de abajo para el caso contrario. Para observar dónde está el error, veamos cómo quedaría el circuito con esta transferencia escrita de esta forma:

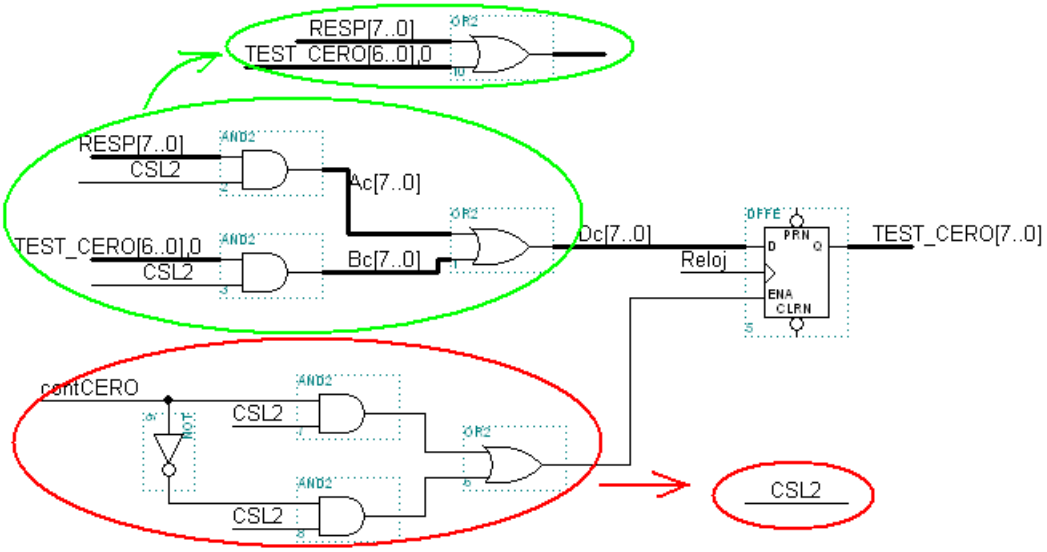


Figura 2.8: Ejemplo de doble transferencia

por lo que podemos observar, en realidad lo que estaríamos haciendo es transferir siempre (Resp OR `Test_cero[6..0],0`), que no es lo que se buscaba. Cabe resaltar que **en RTL está prohibido** designar un mismo registro como destino de distintas transferencias (es decir, a la izquierda de la flecha) en un mismo paso.

Otro error muy común consiste en no tener en cuenta los tiempos del problema. Es importante tener en cuenta que cada paso dura un período, que las transferencias a los registros se realizan al final del paso, mientras que los valores asignados a las salidas son válidos durante el paso. En este problema un error de este tipo podría darse si en el paso 1 no se realizara la transferencia:

```
1. ...
   Test_uno <- Resp
   ...
```

y se agregara un paso 1.5 con esta transferencia:

```
1.5 Test_uno <- Resp
```

En este caso, test ya no valdría 1, por lo que la entrada Resp no correspondería con la información necesaria. Esto se puede ver también en el diagrama de tiempos.

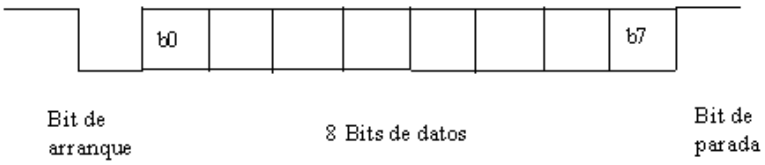
2.3. Examen de Febrero de 1999 - Problema 2

2.3.1. Introducción

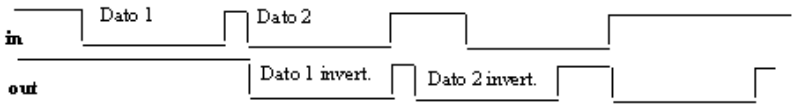
En este ejercicio vamos a ver cómo resolver un problema con dos circuitos RTL interconectados, de modo de simplificar la resolución del mismo.

2.3.2. Letra del problema

Diseñar un circuito RTL que invierta el orden de transmisión de los bits en una comunicación serial. El circuito tendrá una entrada de datos serie in y una salida de datos out. La recepción y trasmisión se hace con un bit de arranque en cero, 8 bits de datos y al menos un bit de parada en uno.
Se supondrá que la señal in cambia sincronizada con una señal de reloj que se utilizará como reloj del sistema. El circuito deberá tener una entrada de RESET.



Se deberá tener en cuenta que la trasmisión de un byte y el siguiente pueden ser contiguas (separados por el bit de parada), con lo cual el sistema podría estar recibiendo un byte y al mismo tiempo trasmitiendo el anteriormente recibido.



Nota: se admitirán soluciones en más de un bloque RTL, siempre que se especifiquen sus conexiones e interfaz.

2.3.3. Solución

Para resolver este problema, tenemos en cuenta la nota, y dividimos el circuito en dos bloques: uno se encarga de la recepción desplazando los bits de entrada **in** en el registro `BUF_IN[8]` y el otro bloque se encarga de la trasmisión, desplazando un registro `BUF_OUT[8]` hacia la salida **out**.

Secuencia RTL del receptor

```
MODULE: RECEPTOR
INPUTS: in
MEMORY: BUF_IN[8], CONT_IN[3]
OUTPUTS: DATO[8], start_out

1. CONT_IN[] <- 7
-> (in, !in) / (1,2)
2. CONT_IN[] <- DEC( CONT_IN[])
BUF_IN[7..0] <- BUF_IN[6..0], in
-> (cinfin, !cinfin) / (3,2)
3. start_out = 1
-> (1)

END SEQUENCE
CONTROL RESET (1)
DATO[] = BUF_IN[]
cinfin = NOT( CONT_IN[2] or CONT_IN[1] or CONT_IN[0] )
END
```

Secuencia RTL del transmisor

```
MODULE: TRANSMISOR
INPUTS: DATO[8], start_out
MEMORY: BUF_OUT[8], CONT_OUT[3]
OUTPUTS: out

1. out=1; CONT_OUT[] <- 7; BUF_OUT[] <- DATO[]
-> (start_out, !start_out) / (2,1)
2. out = 0
3. out = BUF_OUT[0]; CONT_OUT[] <- DEC( CONT_OUT[])
BUF_OUT[7..0] <- 0, BUF_OUT[7..1]
-> (coutfin, !coutfin) / (1,3)

END SEQUENCE
CONTROL RESET (1)
coutfin = !( CONT_OUT[2] or CONT_OUT[1] or CONT_OUT[0] )
END
```

Comentarios

Se puede observar que en este problema, cada módulo por separado es muy sencillo. Básicamente están formados por un shift-register y un contador. El único punto a tener especial atención es que no se pierdan datos en la recepción y en la transmi-

sión, y que la transferencia paralela tiene que realizarse durante un bit de parada o de arranque. Por otra parte, la comunicación en paralelo entre los módulos, y el hecho de que todo sea síncrono con el mismo reloj, hace que el protocolo o la señalización de aviso de dato válido se pueda resolver con una única señal de activación durante 1 período de reloj.

Circuitos

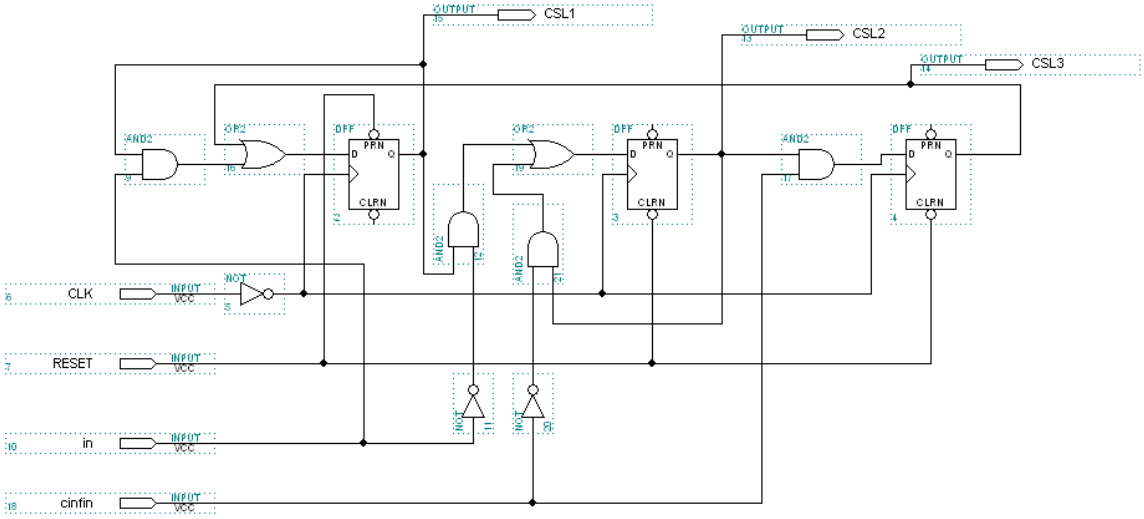


Figura 2.9: Bloque de control del módulo receptor

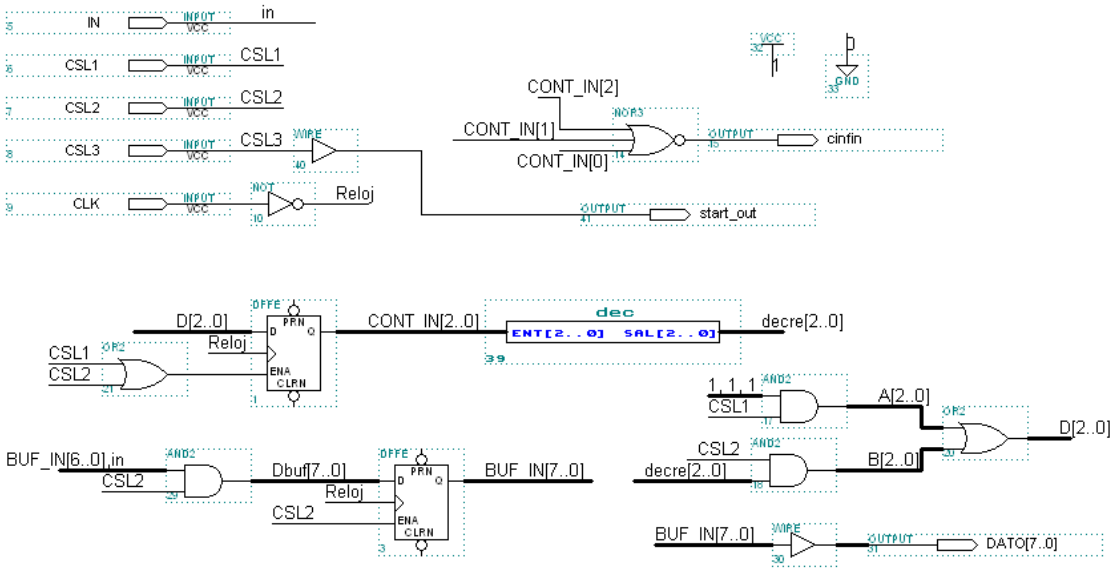


Figura 2.10: Bloque de datos del módulo receptor

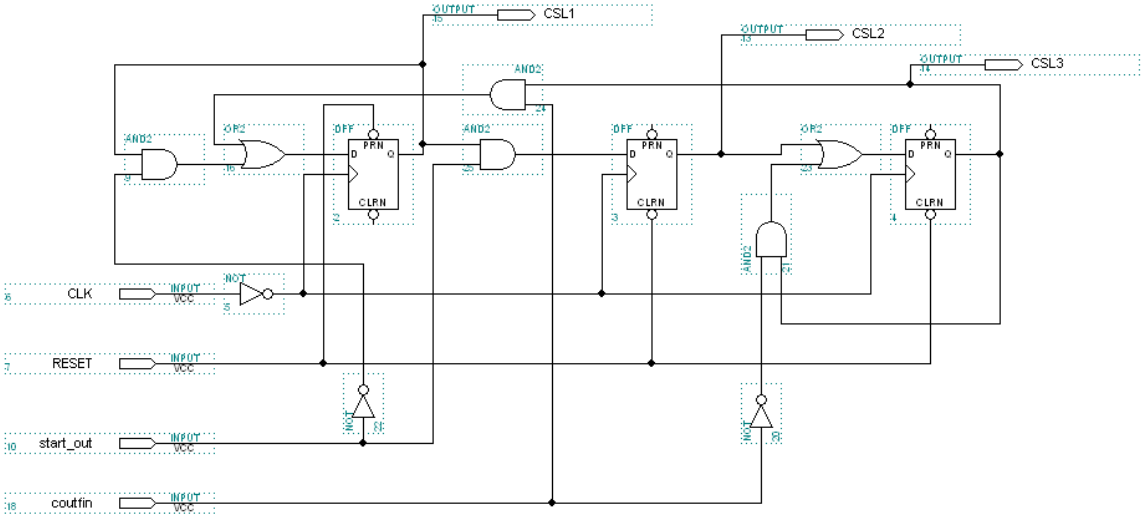


Figura 2.11: Bloque de control del módulo transmisor

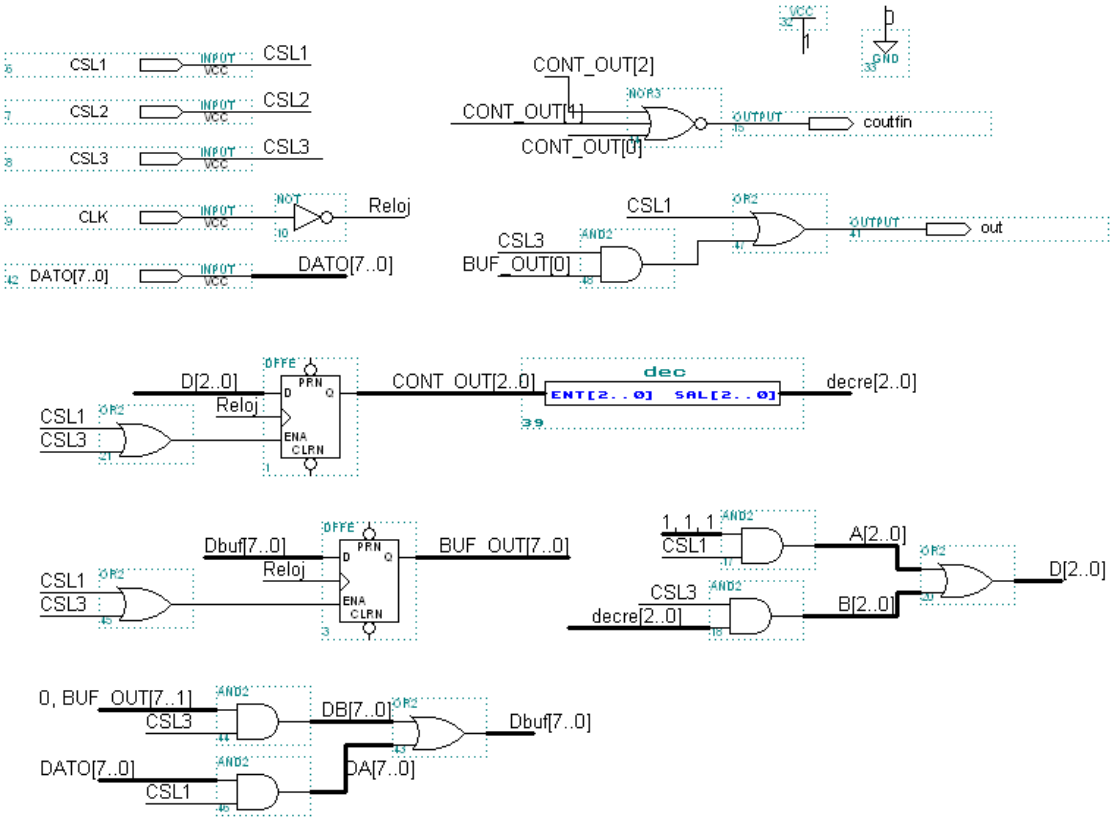


Figura 2.12: Bloque de datos del módulo transmisor

Diagrama de tiempos

La figura 2.14 muestra 3 palabras transmitidas, se indica con una A sobre el diagrama el bit de arranque (nivel bajo) y con una P el de parada (nivel alto). Las primeras dos palabras son consecutivas, es decir que están separadas únicamente con un bit de parada (situación que la letra aclara que hay que considerar). Para la simulación se eligió además mandar otra palabra separada por un número mayor de bits (3).

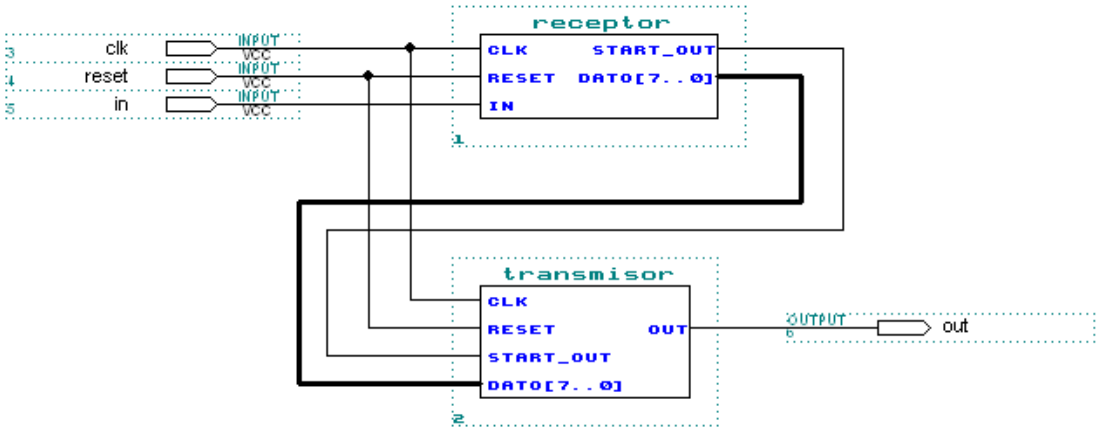


Figura 2.13: Interconexión de bloques

Se verifica que el circuito se comporta según la especificación de la letra para los casos simulados.

Al archivo de simulación adjunto se le pueden agregar las señales CSL para observar en qué pasos se realizan las transferencias.

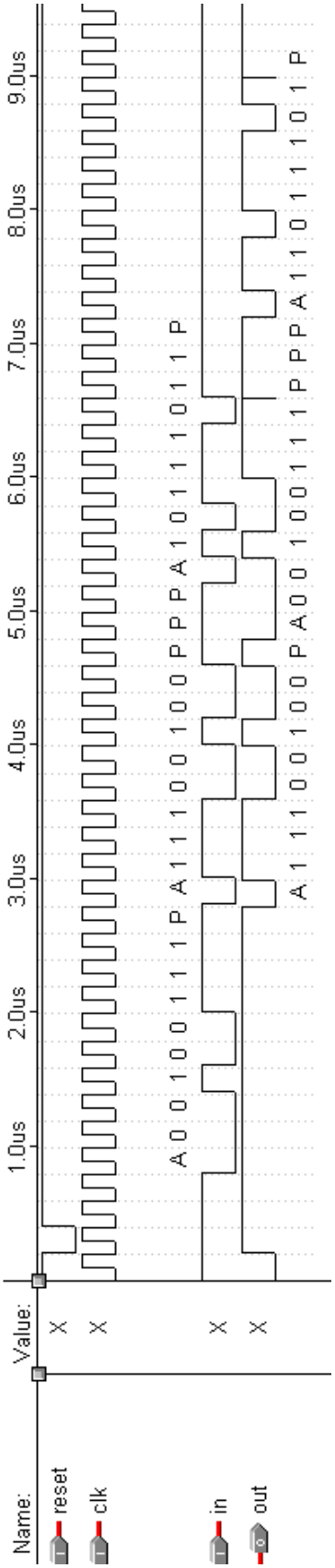


Figura 2.14: Diagrama de tiempos