SEMINARIO DE EXTENSIÓN

Aplicaciones con Lógica Programable en Comunicaciones

Introducción a los Sistemas Lógicos y Digitales

23 de Noviembre del 2007

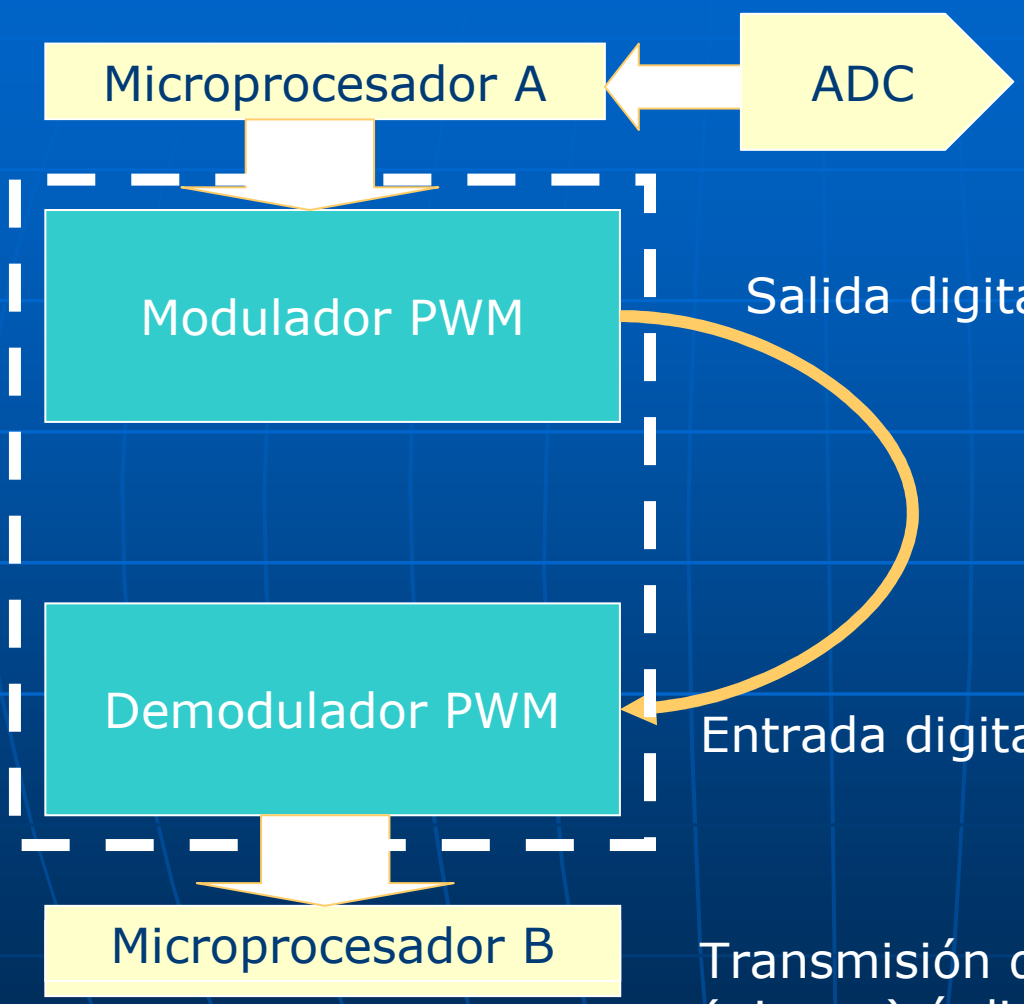
Modulador - Demodulador PWM.

Codificador - Decodificador Manchester.

UART (Transceptor Asincrónico Universal).

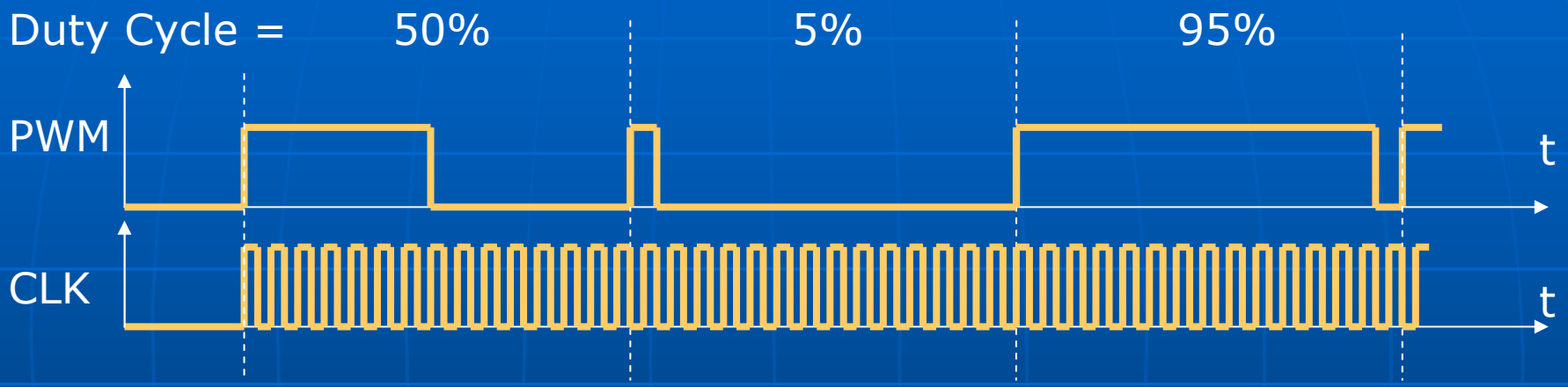
Conclusiones.

Modulador - Demodulador PWM



Transmisión de señales analógicas
(ej. voz) ó digitales (ej. música codificada)

Modulador - Demodulador PWM



Este diseño de PWM es digital.
La resolución la dá el contador (en este caso: 1/1024).
La frecuencia de muestreo máxima la dá el reloj junto con el contador (en este caso: (1/1024)x frec. de reloj).
La entrada al modulador es digital en formato paralelo de 10 bits.
La salida del demodulador es digital en formato paralelo de 10 bits.

Modulador - Demodulador PWM

```
constant MAX_COUNT = 1024;

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_compare.inc";
INCLUDE "lpm_latch.inc";
SUBDESIGN pwm03
(
    reloj_xtal, dato_in[9..0], reset                    : INPUT;
    salidatx_pwm, salida_muestreo, dato_salida_rx[9..0] : OUTPUT;
    reset_rx, zrx, zrrx                                 : OUTPUT;
)
VARIABLE
    contadortx: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    contadorrx: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    comparador: lpm_compare WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    entradatx: lpm_latch WITH (LPM_WIDTH=10);
    salidarx:  lpm_latch WITH (LPM_WIDTH=10);
    salida_pwm : DFF;
    muestreo : DFF;
    entradarx_pwm, yrx, yrrx : NODE;


                %   current   current %
                %   state     output  %

 ssrx: MACHINE OF BITS (zrx)
       WITH STATES (s0    =   0,
                    s1    =   0,
                    s2    =   1,
                    s3    =   0);

                %   current   current %
                %   state     output  %

 ssrrx: MACHINE OF BITS (zrrx)
       WITH STATES (s0r    =   0,
                    s1r    =   0,
                    s2r    =   1,
                    s3r    =   0);
```

DECLARACIONES

Declaraciones de entrada-salida

Declaraciones de componentes parametrizados

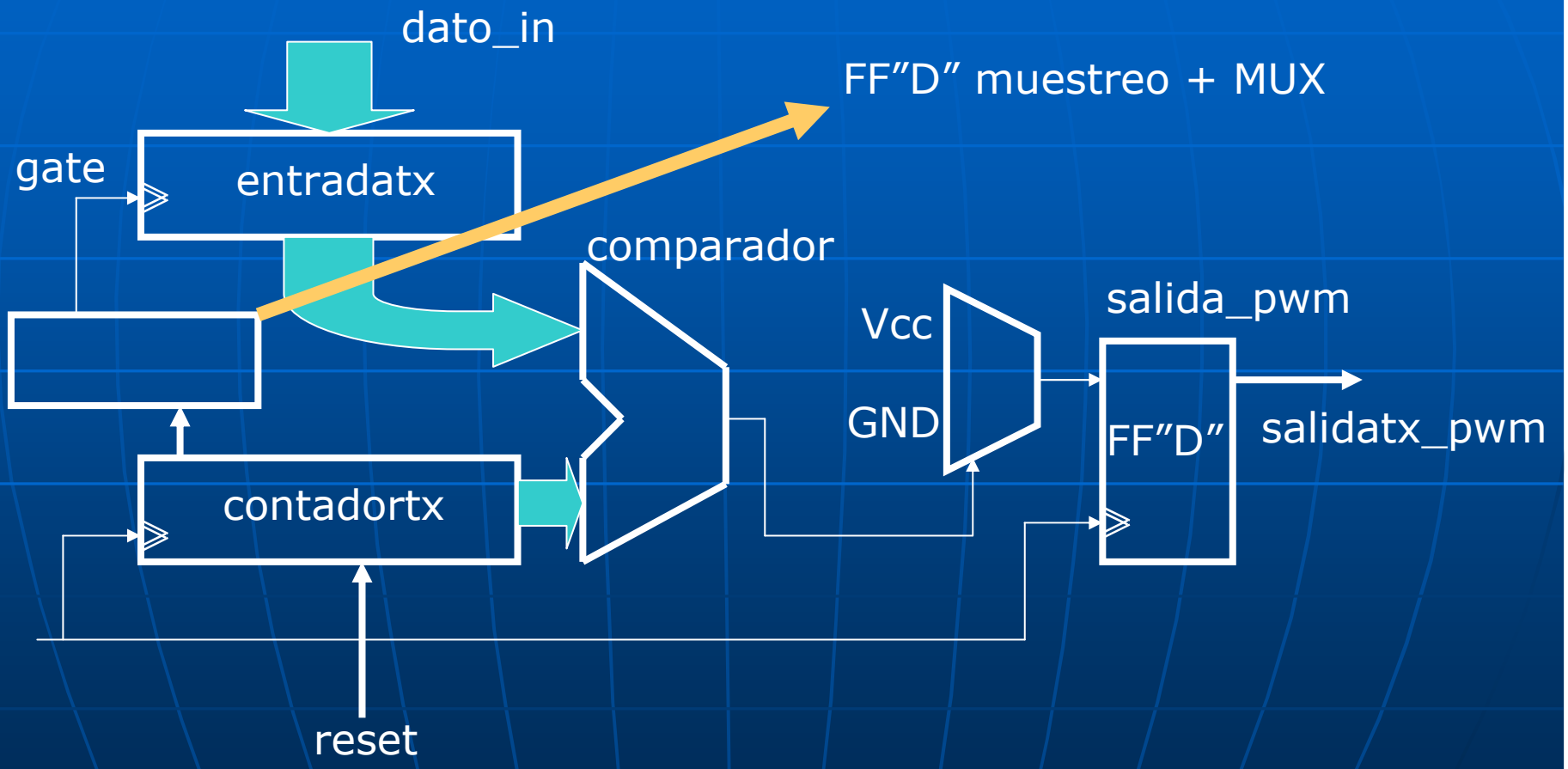
Declaraciones de variables internas

Máquina de Moore en RX para generar pulso de carga para copia de datos desde contador.

Máquina de Moore en RX para generar pulso de reset del contador

Modulador - Demodulador PWM

ESQUEMA DEL MODULADOR



Modulador - Demodulador PWM

```
BEGIN
    entradatx.data[] = dato_in[];
    muestreo.clk = reloj_xtal;
    salida_pwm.clk = reloj_xtal;
    contadortx.clock = reloj_xtal;
    contadortx.sclr = reset;
    comparador.dataa[] = entradatx.q[];
    comparador.datab[] = contadortx.q[];
            IF comparador.aleb == 0 THEN
                salida_pwm.d = VCC;
            ELSE
                salida_pwm.d = GND;
            END IF;
 salidatx_pwm = salida_pwm.q;

            IF contadortx.eq[0] == 1 THEN
                muestreo.d = VCC;
            ELSE
                muestreo.d = GND;
            END IF;

    entradatx.gate = muestreo.q;
    salida_muestreo = muestreo.q;
```

ETAPA DEL MODULADOR

entradtx = latch de 10 bits para carga de dato al modulador

salida_pwm = FF “D” para generar señal PWM

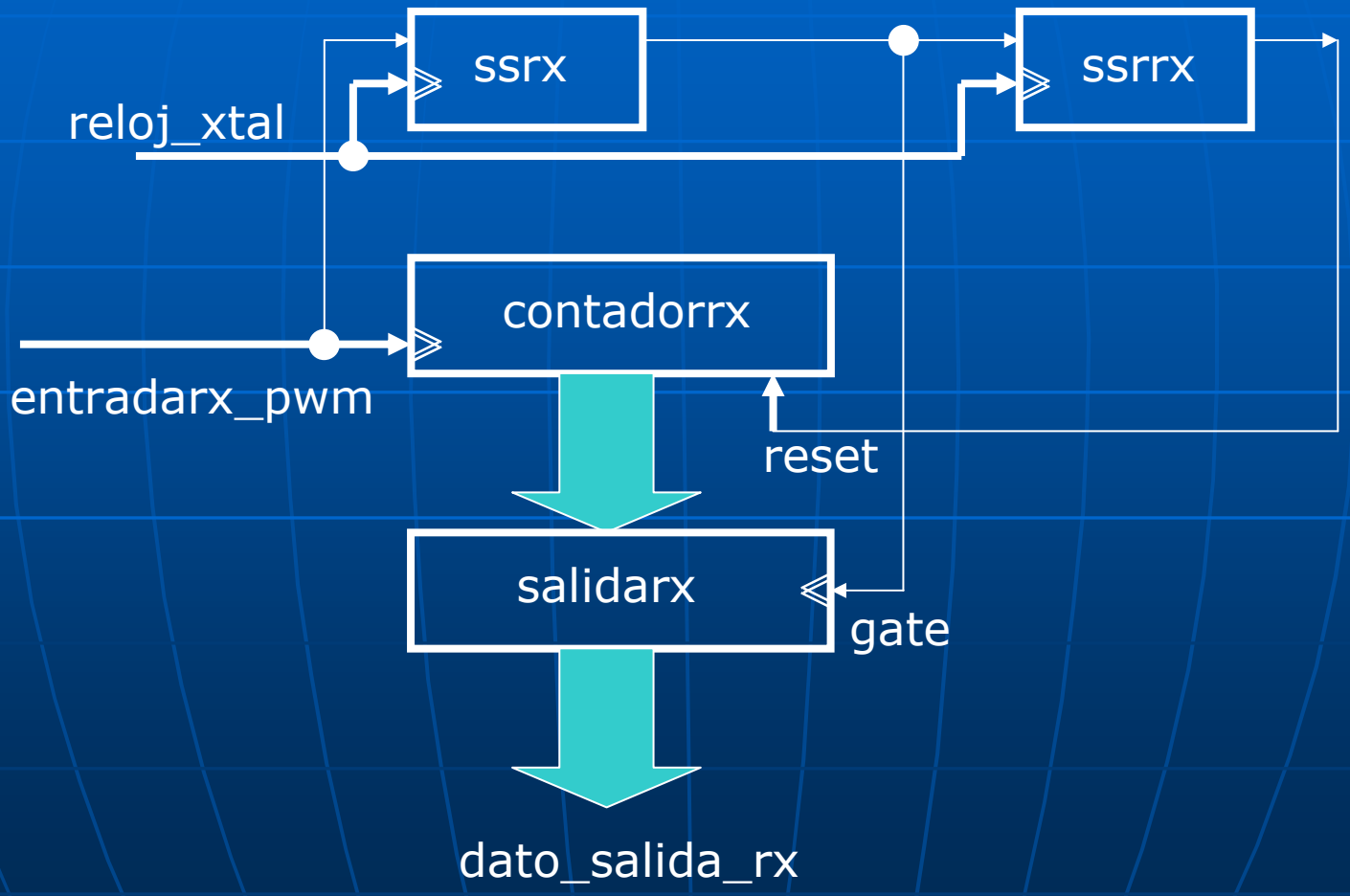
muestreo = FF “D” para generar pulso de actualización de carga de dato de entrada

contadortx = contador para modulador

comparador = comparador entre el valor de entrada y contador

Modulador - Demodulador PWM

ESQUEMA DEL DEMODULADOR



Modulador - Demodulador PWM

```
    entradarx_pwm = salidatx_pwm;
    contadorrx.clock = reloj_xtal;
    reset_rx = zrrx;
    contadorrx.aclr = reset # reset_rx;
    contadorrx.cnt_en = entradarx_pwm;
    salidarx.data[] = contadorrx.q[];
    salidarx.gate = zrx;
    dato_salida_rx[] = salidarx.q[];
    yrx = entradarx_pwm;

  TABLE
  %  estado   entrada      estado %
  %  actual   actual       próximo%
     ssrx,  yrx       =>    ssrx;

     s0,       0       =>    s0;
     s0,       1       =>    s1;
     s1,       0       =>    s2;
     s1,       1       =>    s1;
     s2,       0       =>    s3;
     s2,       1       =>    s3;
     s3,       0       =>    s3;
     s3,       1       =>    s1;
  END TABLE;
```

ETAPA DEL DEMODULADOR

contadorrx = Parte de RX para la medición de pulsos de reloj en T1

salidarx = latch para actualización de datos en port paralelo

Máquina de Moore para “carga” de datos del contadorrx a salida de latch “salidarx”

Modulador - Demodulador PWM

ETAPA DEL DEMODULADOR

```
    TABLE
    %  estado   entrada      estado %
    %  actual   actual       próximo%
       ssrrx,   yrrx      =>   ssrrx;

       s0r,       0       =>   s0r;
       s0r,       1       =>   s1r;
       s1r,       0       =>   s2r;
       s1r,       1       =>   s1r;
       s2r,       0       =>   s3r;
       s2r,       1       =>   s3r;
       s3r,       0       =>   s3r;
       s3r,       1       =>   s1r;
    END TABLE;

     yrrx = zrx;
     ssrx.clk = reloj_xtal;
     ssrx.reset = reset;

     ssrrx.clk = reloj_xtal;
     ssrrx.reset = reset;
END;
```

Máquina de Moore para reset de contadorrx

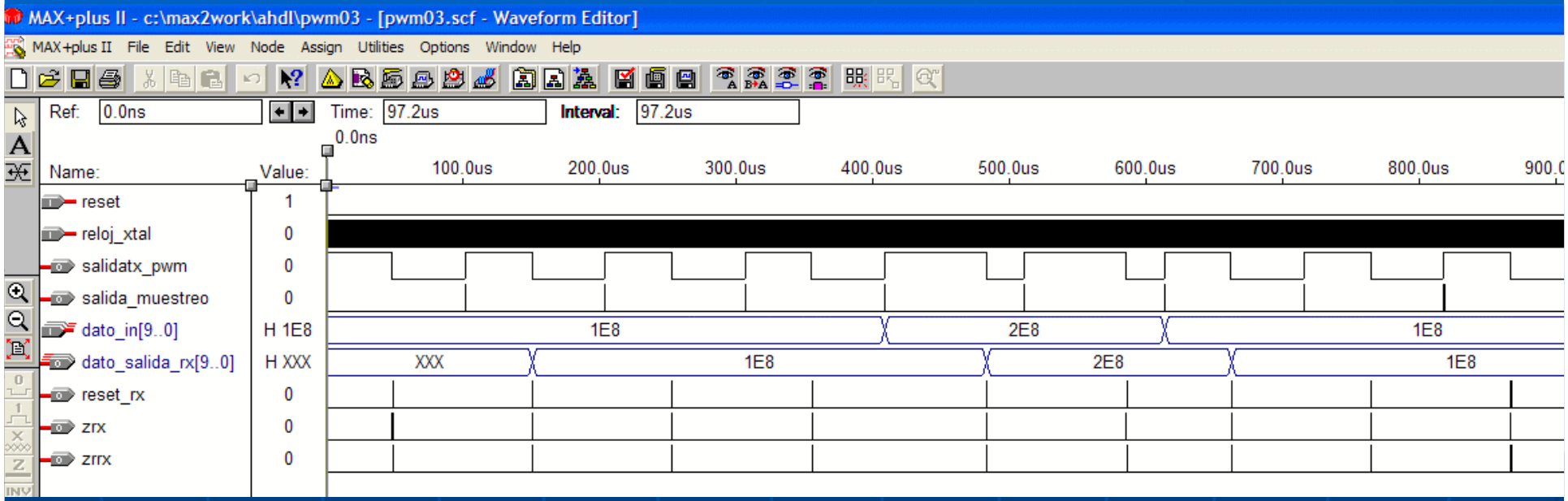
Señales de máquina de “carga”

Señales de máquina de “reset”

NOTA: Por simplicidad para los fines de la simulación “reloj_xtal” es el mismo en TX y RX.

Modulador - Demodulador PWM

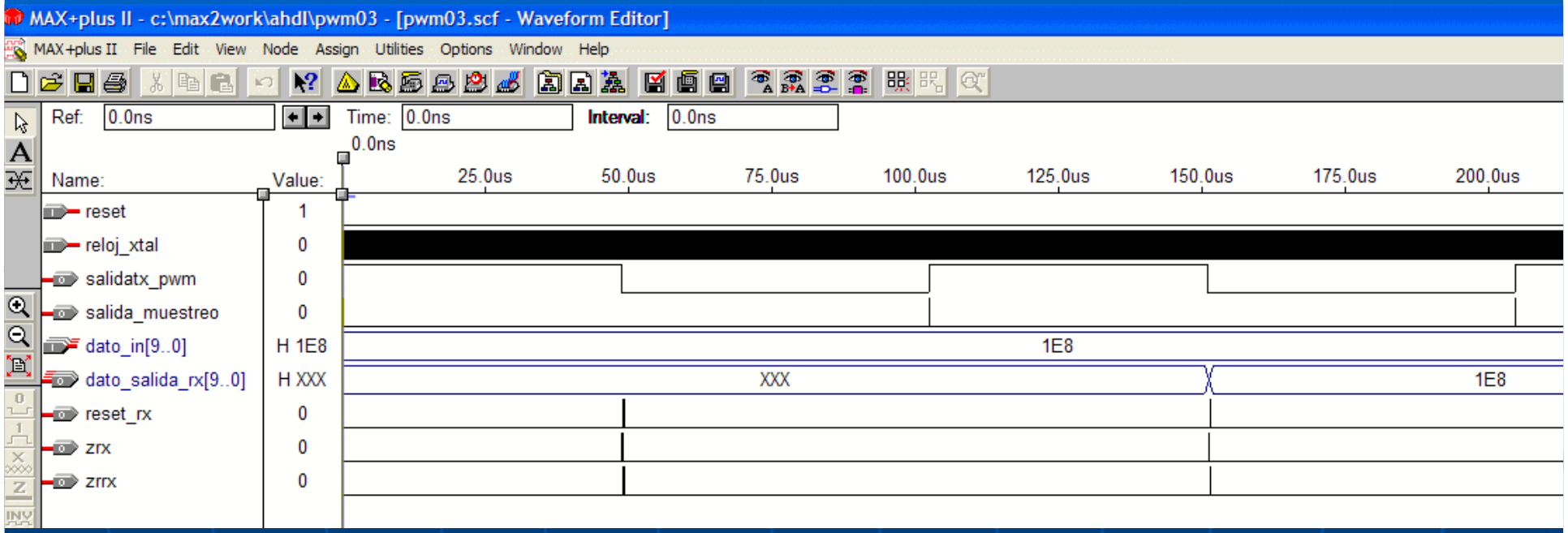
SIMULACIÓN DE MODULADOR Y DEMODULADOR



Nota: Aquí se simula con el mismo reloj “reloj_xtal”.
Se supone que en la realidad los relojes del TX y RX deben ser asincronicos uno del otro.

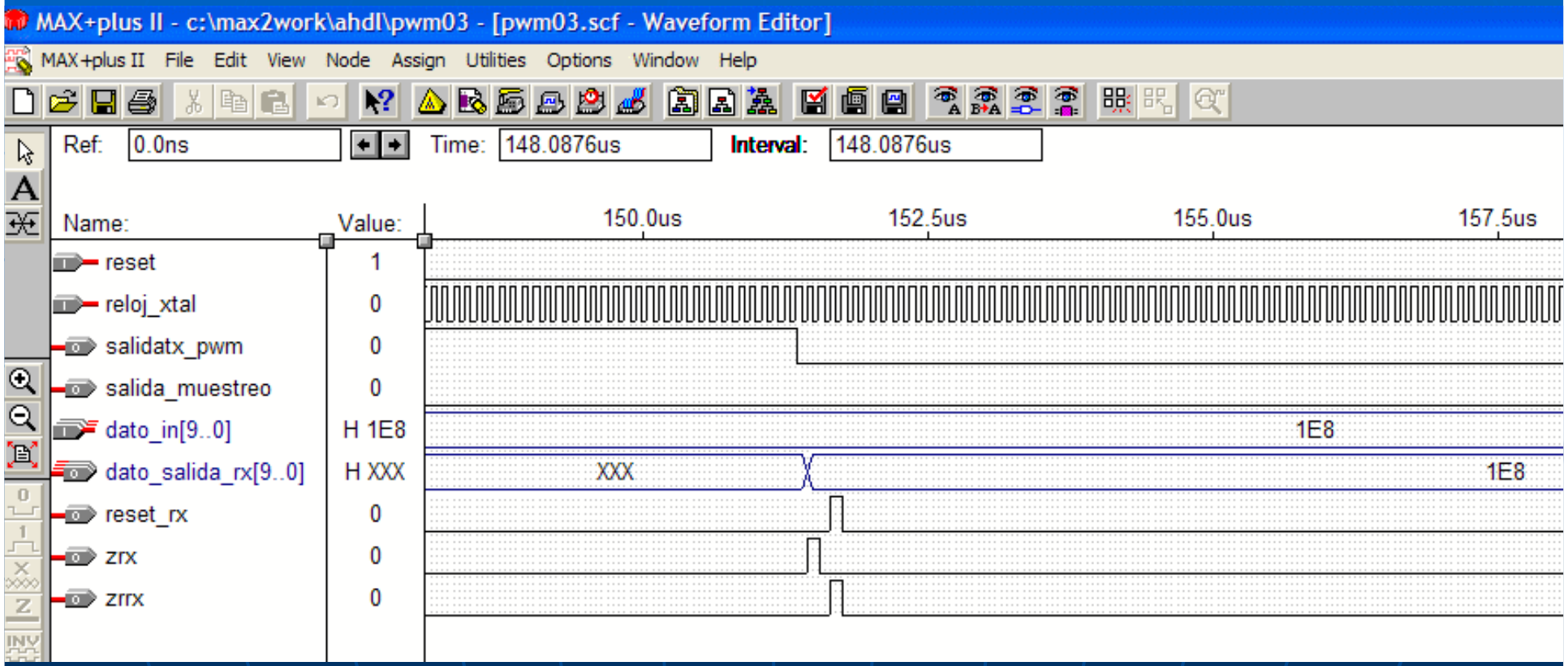
Modulador - Demodulador PWM

SIMULACIÓN DE MODULADOR Y DEMODULADOR

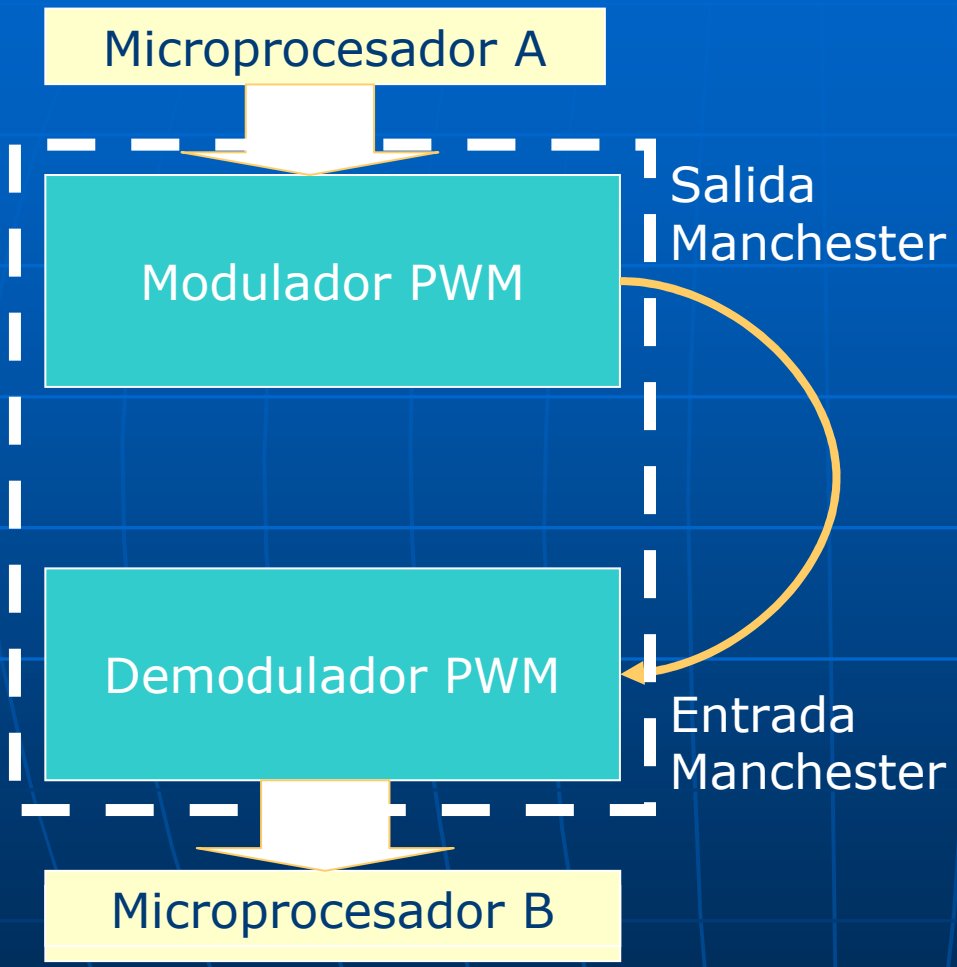


Modulador - Demodulador PWM

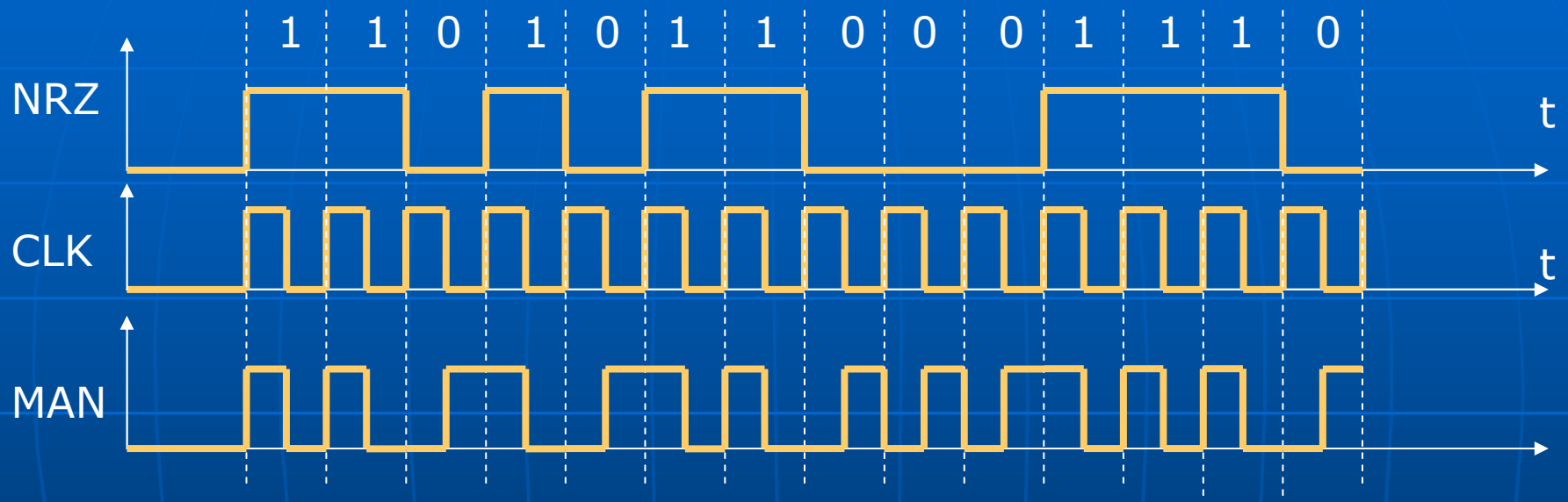
SIMULACIÓN DE MODULADOR Y DEMODULADOR



Codificador - Decodificador Manchester



Código de línea bifásico autosincronizable. Empleado en Ethernet 10basexx.



El código Manchester es un esquema de modulación PSK donde la señal portadora es una onda cuadrada y se modula en fase con 0º ó 180º dependiendo del dato binario (0 ó 1 respectivamente).
Es autosincronizable: se puede rescatar la información de reloj ya que siempre existe al menos una transición del dato en un tiempo de bit.

Codificador - Decodificador Manchester

```
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_shiftreg.inc";
INCLUDE "lpm_latch.inc";        % gate=1 habilita, gate=0 latch %

SUBDESIGN codec_man02
(
    reset, relojx16, datoentrada[7..0]              : INPUT;
    relojx2, relojx1, dato_salidatx                 : OUTPUT;
    dato_salidarx, flanco, salida_contador[3..0]    : OUTPUT;
    load, dato_salidarxparalelo[7..0]               : OUTPUT;
)
VARIABLE
    prescaler: lpm_counter WITH ( LPM_WIDTH=4);
    contadortx: lpm_counter WITH ( LPM_WIDTH=4);
    contadorrx: lpm_counter WITH ( LPM_WIDTH=4);
    desplaza: lpm_shiftreg WITH ( LPM_WIDTH=16);
    detector_rx: lpm_shiftreg WITH ( LPM_WIDTH=16);
    latchx8_rx: lpm_latch WITH ( LPM_WIDTH=8);
    label[7..0], carga, dato_entradarx              :   NODE;
    ytx, dato_entradatx, yrx                        :   NODE;
    salida      : DFF;

                %   estado   salida %
                %   actual   actual %
   sstx: MACHINE OF BITS (ztx)
       WITH STATES (s0t    =   0,
                    s1t    =   1,
                    s2t    =   0,
                    s3t    =   0,
                    s4t    =   1,
                    s5t    =   0,
                    s6t    =   0,
                    s7t    =   0);

                %   estado   salida %
                %   actual   actual %
 ssrx: MACHINE OF BITS (zrx)
       WITH STATES (s0    =   0,
                    s1    =   1,
                    s2    =   0,
                    s3    =   1);
```

DECLARACIONES

Declaraciones de entrada-salida

Declaraciones de componentes parametrizados

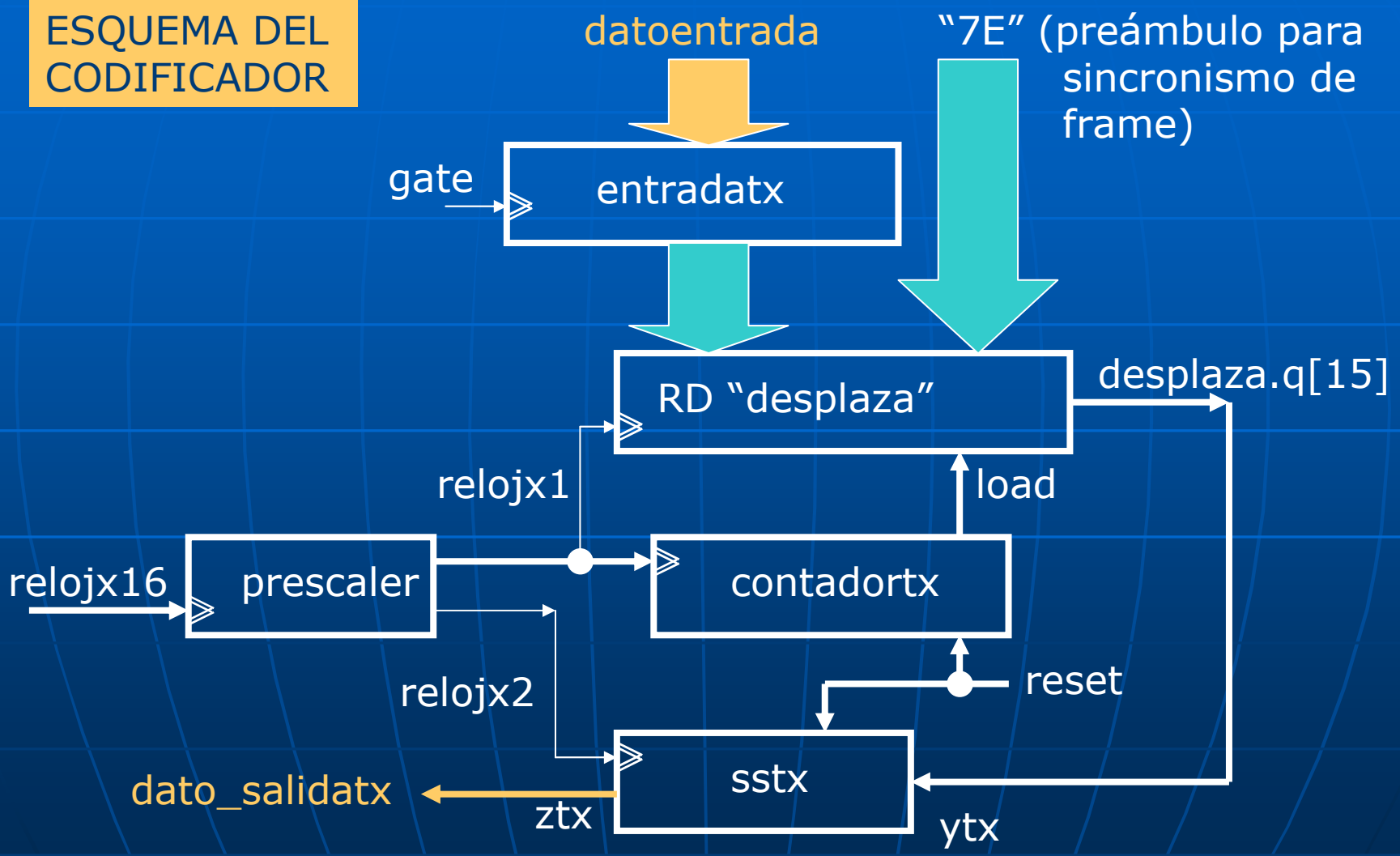
Declaraciones de variables internas

Máquina de Moore en TX para generar salida manchester con reloj de doble cadencia (relojx2).

Máquina de Moore en RX para generar pulso de detección de flancos

Codificador - Decodificador Manchester

ESQUEMA DEL CODIFICADOR



Codificador - Decodificador Manchester

```
BEGIN
%   Comienzo de etapa del modulador Manchester  %
    label[] = H"7E";
    contadortx.aclr = reset;
    desplaza.aclr = reset;
    prescaler.aclr = reset;
    prescaler.clock = relojx16;
    relojx2 = prescaler.q[2];
    relojx1 = prescaler.q[3];
    contadortx.clock =  relojx1;
    desplaza.clock = relojx1;   % Reloj del shift register: T=2us %
    desplaza.data[15..8] = label[];
    desplaza.data[7..0] = datoentrada[];
    dato_entradatx = desplaza.q[15];
    IF contadortx.q[] == H"0" THEN
        desplaza.load = VCC;
        carga = VCC;
    ELSE
        desplaza.load = GND;
        carga = GND;
    END IF;

    IF contadortx.q[] == H"0" & !relojx1 THEN
        load = VCC;
    ELSE
        load = GND;
    END IF;

    ytx = dato_entradatx;
    dato_salidatx = ztx;
   sstx.clk   = relojx2;
   sstx.reset = reset;
```

Codificador - Decodificador Manchester

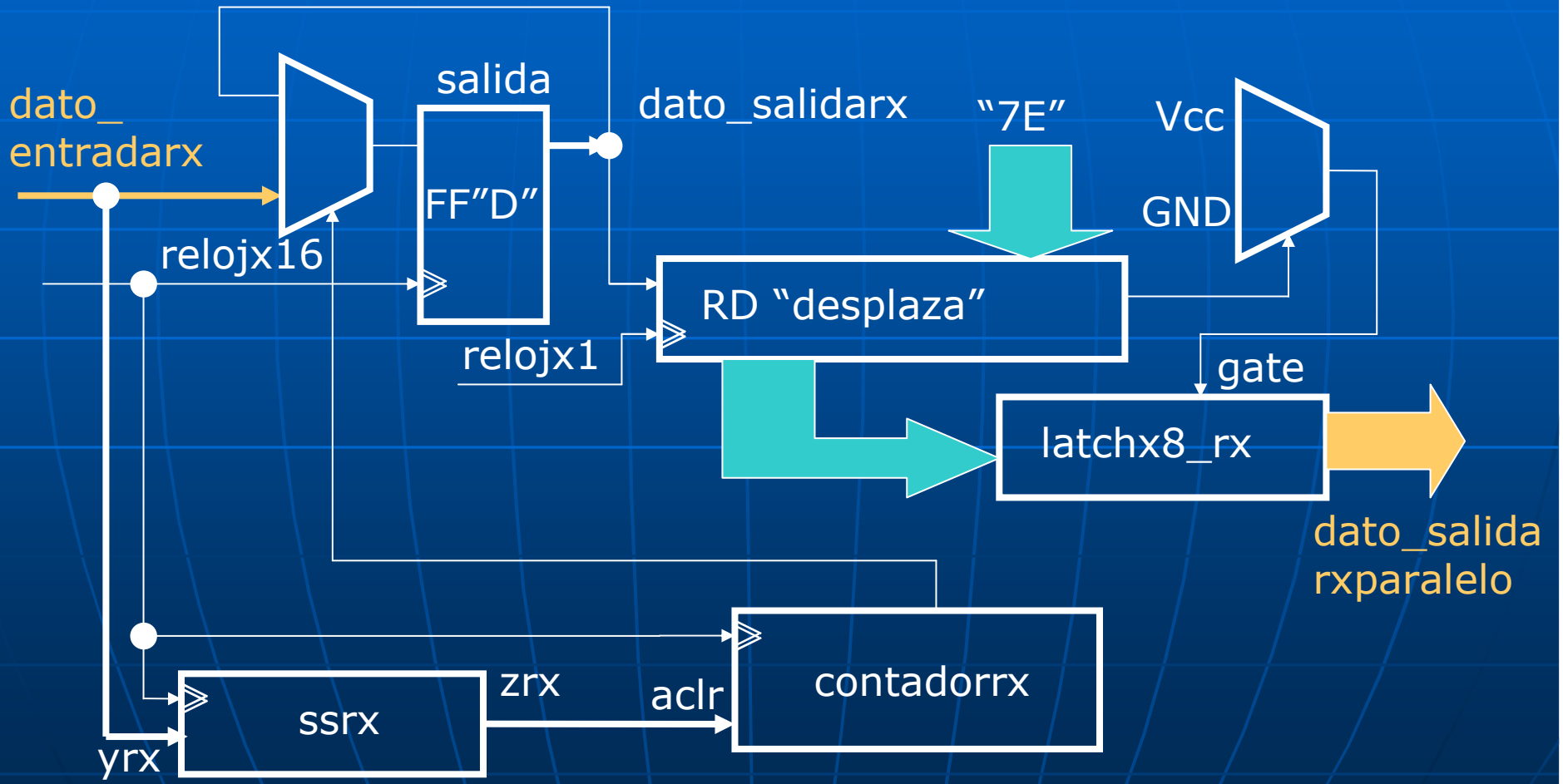
```
TABLE
%  estado  entrada      estado%
%  actual  actual       próximo%
   sstx,      ytx      =>   sstx;
   s0t,       0        =>   s0t;
   s0t,       1        =>   s1t;
   s1t,       0        =>   s2t;
   s1t,       1        =>   s2t;
   s2t,       0        =>   s3t;
   s2t,       1        =>   s1t;
   s3t,       0        =>   s4t;
   s3t,       1        =>   s4t;
   s4t,       0        =>   s3t;
   s4t,       1        =>   s1t;
END TABLE;

%   Comienzo de etapa del demodulador Manchester    %

ssrx.clk   = relojx16;
ssrx.reset = reset;

 dato_entradarx = dato_salidatx;

 yrx = dato_entradarx;
 flanco = zrx;

TABLE
%  estado  entrada      estado%
%  actual  actual       próximo%
   ssrx,      yrx      =>   ssrx;
   s0,        0        =>   s0;
   s0,        1        =>   s1;
   s1,        0        =>   s2;
   s1,        1        =>   s2;
   s2,        0        =>   s3;
   s2,        1        =>   s2;
   s3,        0        =>   s0;
   s3,        1        =>   s0;
END TABLE;
```

Máquina de Moore del TX para generación de código Manchester (trabaja a doble frecuencia que el reloj de datos)

Máquina de Moore del RX para la detección de flancos en la entrada de datos al receptor

Codificador - Decodificador Manchester

ESQUEMA DEL DECODIFICADOR



Codificador - Decodificador Manchester

```
    salida.clk = relojx16;
    dato_salidarx = salida.q;

    contadorrx.clock =  relojx16;
    contadorrx.aclr = flanco # reset;
    salida_contador[] = contadorrx.q[];
    dato_salidarx = salida.q;
    CASE contadorrx.q[] IS
    WHEN 12 =>
        salida.d = dato_entradarx;
    WHEN OTHERS =>
        salida.d = salida.q;
    END CASE;

    detector_rx.shiftin = dato_salidarx;
    detector_rx.clock = relojx1;
    latchx8_rx.data = detector_rx.q[7..0];
    dato_salidarxparalelo[] = latchx8_rx.q[];

    IF detector_rx.q[15..8] == H"7E" THEN
        latchx8_rx.gate = VCC; % sensa %
    ELSE
        latchx8_rx.gate = GND; % congela %
    END IF;

END;
```

Contador “contadorrx” cuenta desde “0” cada vez que la máquina de estados ssrx detecta un flanco en la entrada de datos.
Si el conteo llega a “12” significa que hay una transición entre “0” a “1” ó viceversa.

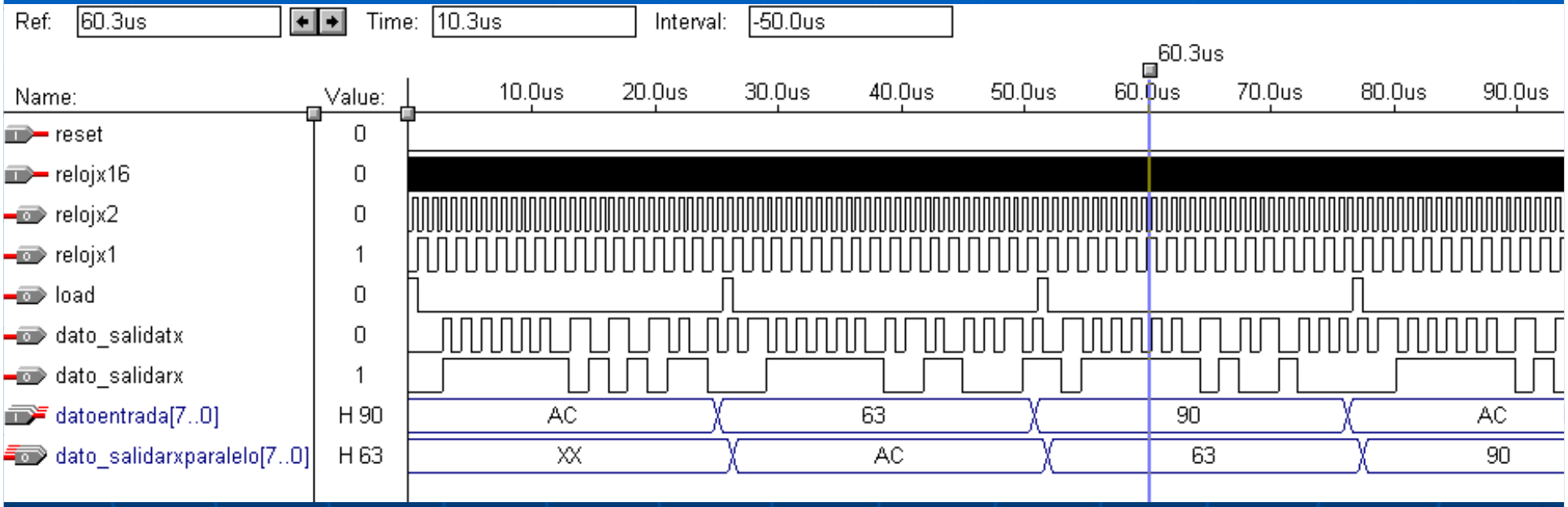
El muestreo del dato Manchester se hace al detectar ese número.

“detector_rx” es un RD que detecta la secuencia “01111110” (“7E”).
Al encontrarla, se refresca la información del “latchx8” el cual recupera el dato en formato paralelo desde el RD.

NOTA: Por simplicidad no se volvió a generar relojx16 y relojx1 en RX ya que se supone que es un CODEC que está en un lazo cerrado.

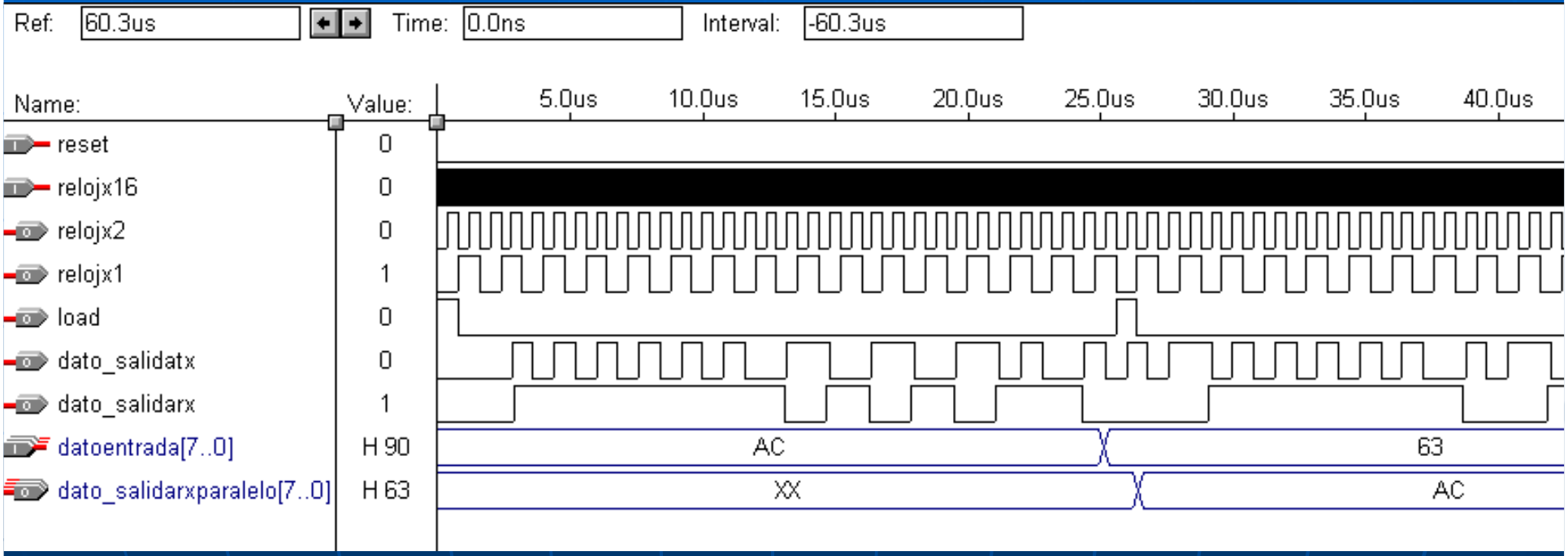
Codificador - Decodificador Manchester

SIMULACIÓN DE CODIFICADOR Y DECODIFICADOR



Codificador - Decodificador Manchester

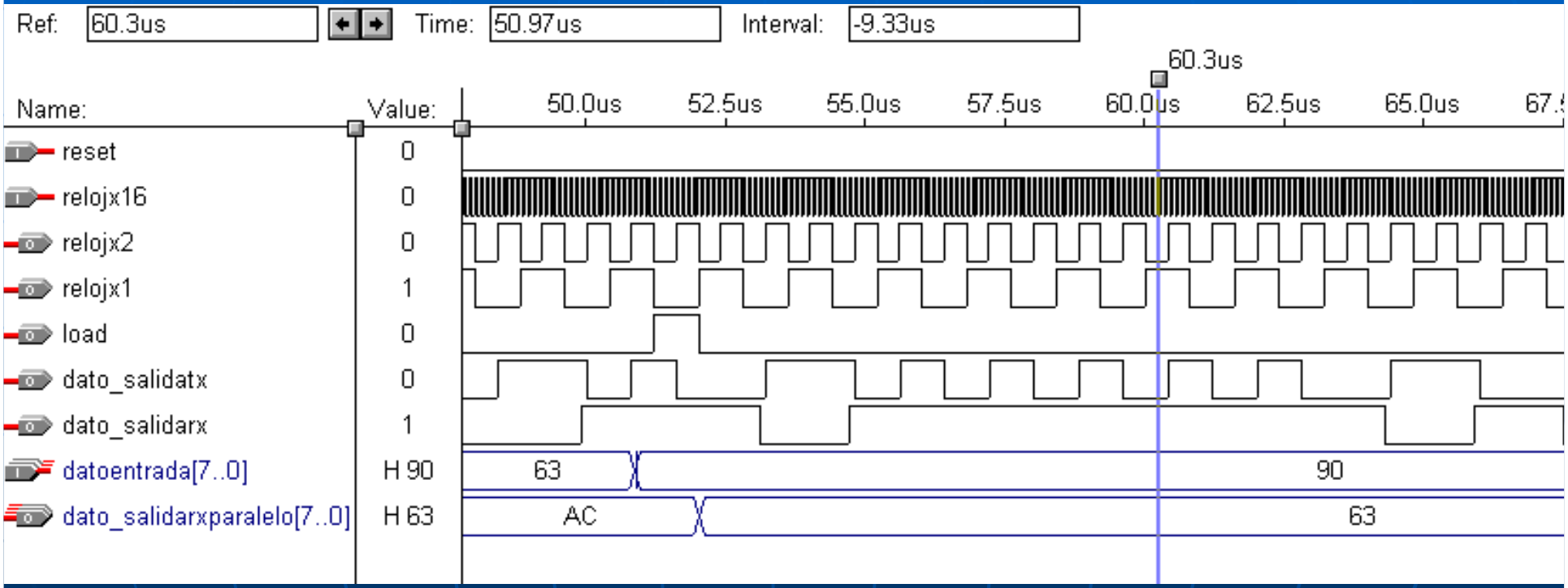
SIMULACIÓN DE CODIFICADOR Y DECODIFICADOR



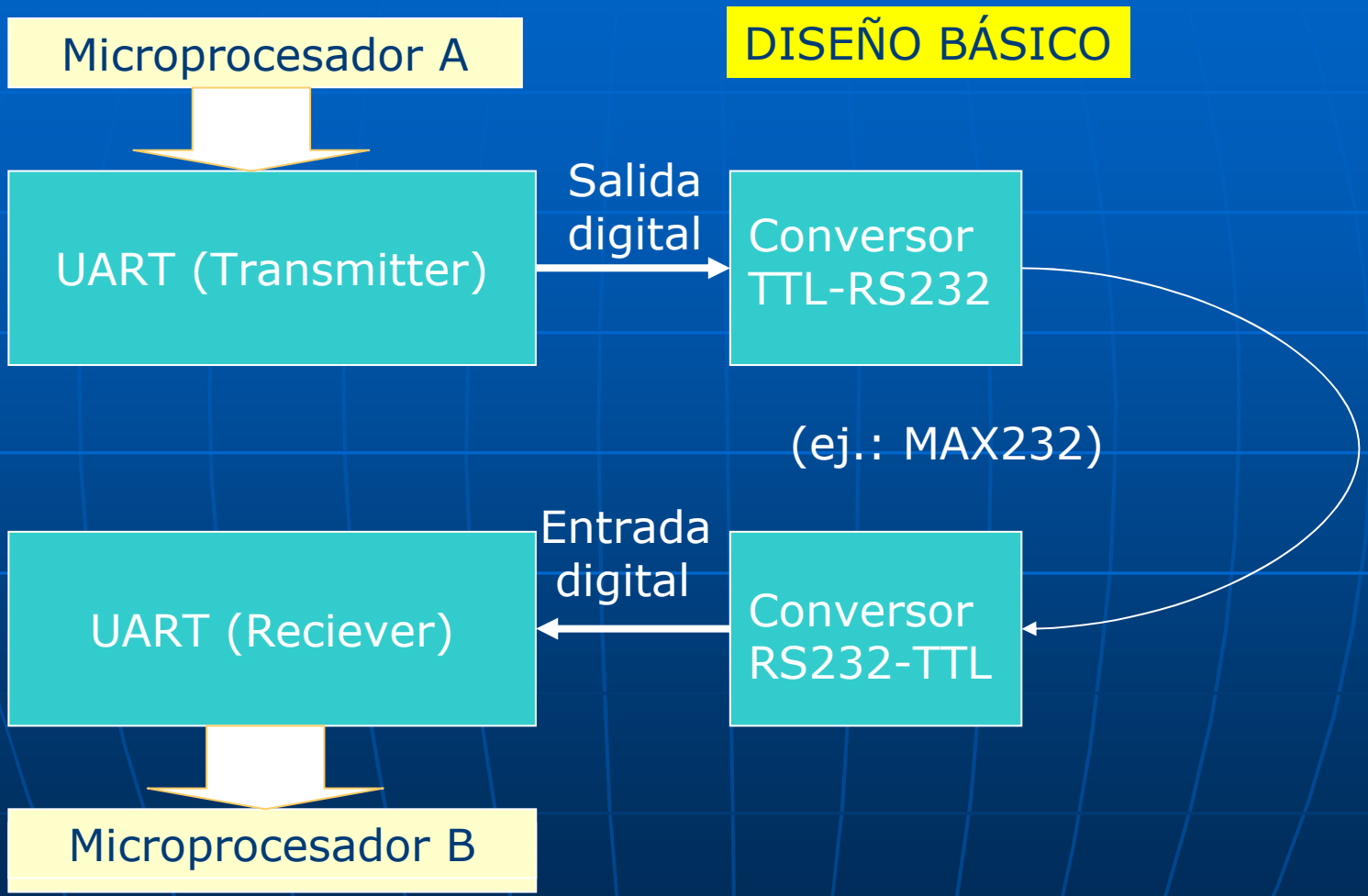
Aplicaciones con Lógica Programable en Comunicaciones

Codificador - Decodificador Manchester

SIMULACIÓN DE CODIFICADOR Y DECODIFICADOR



UART (Universal Asynchronous Receiver-Transmitter)



UART (Universal Asynchronous Receiver-Transmitter)

DISEÑO DEL TRANSMISOR Y RECEPTOR

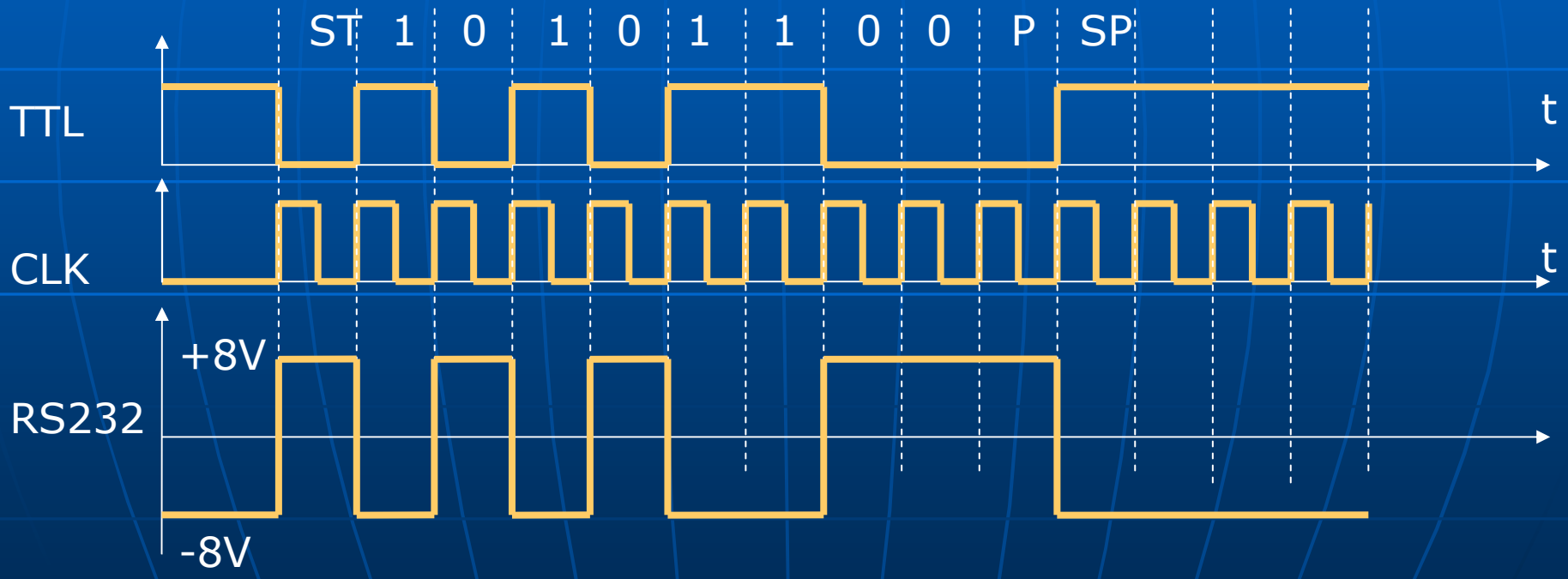
El TX es el mismo que antes salvo cambio de algunos nombres.
El diseño del receptor se basa en un RD de 12 bits donde el circuito está a la espera de detectar el bit de START (transición de 1 a 0).
Al detectarla comineza a funcionar un reloj para ingresar la información a un RD. Dicho reloj "pulsorx" genera 11 pulsos hasta que entren todos los datos.
Al finalizar la carga se convierte la parte de datos en formato paralelo en "darx[]" la cual puede capturar el microprocesador.
Una señal "listo" avisa al mismo cuando puede realizar la operación de lectura.

NOTA: Por simplicidad, no se ha implementado el detector de error de paridad y de frame. Antes de cargar el dato obtenido desde el RD se deben verificar dos cosas:
1) Que se haya detectado el bit de Stop.
2) Que el bit de paridad coincida con la paridad de los datos obtenidos.

UART (Universal Asynchronous Receiver-Transmitter)

DISEÑO DEL TRANSMISOR

"10101100" = "AC" (dato a transmitir)
ST = Bit de START
SP = Bit de STOP
P = Bit de paridad (impar en el ejemplo)



UART (Universal Asynchronous Receiver-Transmitter)

DISEÑO DEL TRANSMISOR

Especificaciones:
1 bit de Start.
1 bit de Stop.
8 bits de datos (MSB primero)
1 bit de Paridad Impar (vale "1" si número de "1´s" en dato es impar).
1 bit de Stop.

El diseño se basa en un RD de 12 bits donde las transiciones de bit de Start y Stop se generan con el mismo RD.

El dato de entrada "dat[]" está en formato paralelo y se carga en TX con la entrada "carga".
La salida "ocupado" avisa al microprocesador si todavía hay una transmisión de datos en curso poniéndose a "1" (buffer no vacío).

UART

DISEÑO DEL TRANSMISOR

```
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_shiftreg.inc";
INCLUDE "lpm_latch.inc";        % gate=1 habilita, gate=0 latch %

SUBDESIGN uart_01
(
    reset, relojx16, dat[7..0], carga                : INPUT;
    relojx2, relojx1, dato_salidatx, carga_desplaza  : OUTPUT;
    paridad, ocupado                                 : OUTPUT;
)
VARIABLE
    prescaler: lpm_counter WITH ( LPM_WIDTH=4);
    contadortx: lpm_counter WITH ( LPM_WIDTH=4);
    desplaza: lpm_shiftreg WITH ( LPM_WIDTH=12);
    habilita            : NODE;
BEGIN
    paridad = dat[0] $ dat[1] $ dat[2] $ dat[3] $ dat[4] $ dat[5] $ dat[6] $ dat[7];
    %Función Or-Exclusiva de 8 variables%

    desplaza.clock = relojx1;
    desplaza.data[11] = VCC;
    desplaza.data[10] = GND;
    desplaza.data[9..2] = dat[7..0];
    desplaza.data[1] = paridad;
    desplaza.data[0] = VCC;
    desplaza.enable = habilita;
    desplaza.load = carga_desplaza;
    prescaler.aclr = reset;
    prescaler.clock = relojx16;
    relojx2 = prescaler.q[2];
    relojx1 = prescaler.q[3];
```

La generación del bit de paridad se hace con una Or-Exclusiva.
"desplaza" es un registro de desplazamiento de 12 bits manejado por el clk "relojx1".
"contadortx" se encarga de contar el número de bits que manda el RD "desplaza".

UART

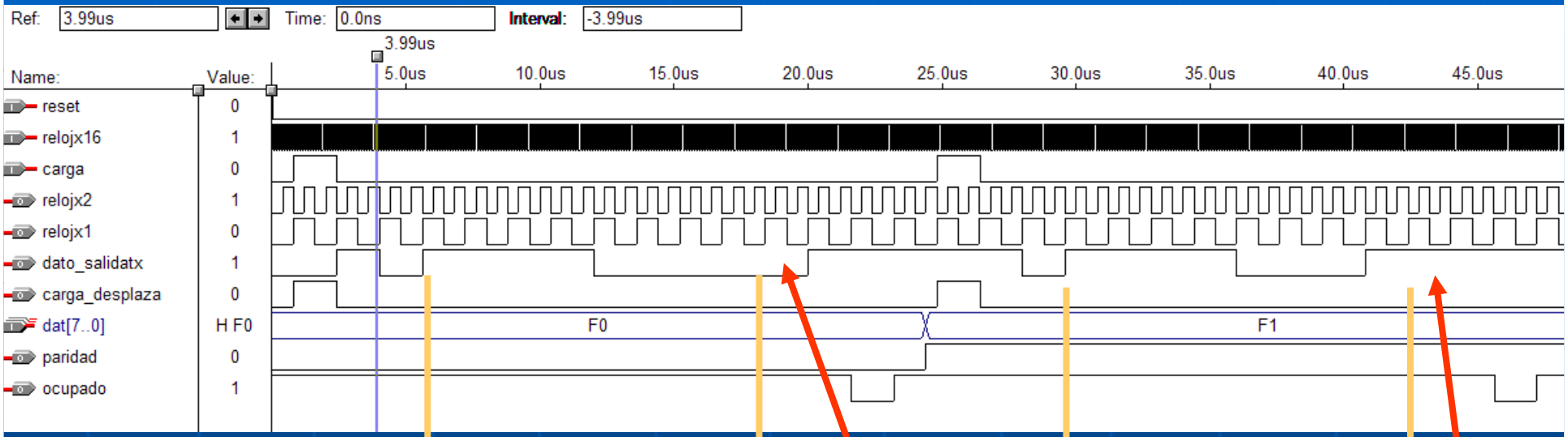
DISEÑO DEL TRANSMISOR

```
    contadortx.aclr = reset;
    contadortx.clock =  relojx1;
    dato_salidatx = desplaza.shiftout;

    IF carga == VCC THEN
        carga_desplaza = VCC;
        contadortx.aclr = VCC;
    ELSE
        carga_desplaza = GND;
    END IF;

    IF contadortx.eq[12] == VCC THEN
        habilita = GND;
    ELSE
        habilita = VCC;
    END IF;

    ocupado = habilita;

END;
```

Cuando se quiere cargar un dato, se levanta la línea "carga" lo que actualiza los datos en el RD.
Esta acción además resetea al contador y habilita al RD a que empiece a sacar datos por "dato_salidatx" hasta que el contador llegue a "12", cuando queda nuevamente inhibido el funcionamiento del RD.
La salida "ocupado" pasa de 1 a 0 durante un ciclo de "reojx1" para avisar al uP que puede volver a cargar un nuevo dato.

SIMULACIÓN DE TRANSMISOR DE LA UART

UART



dato "F0"

P = "0"

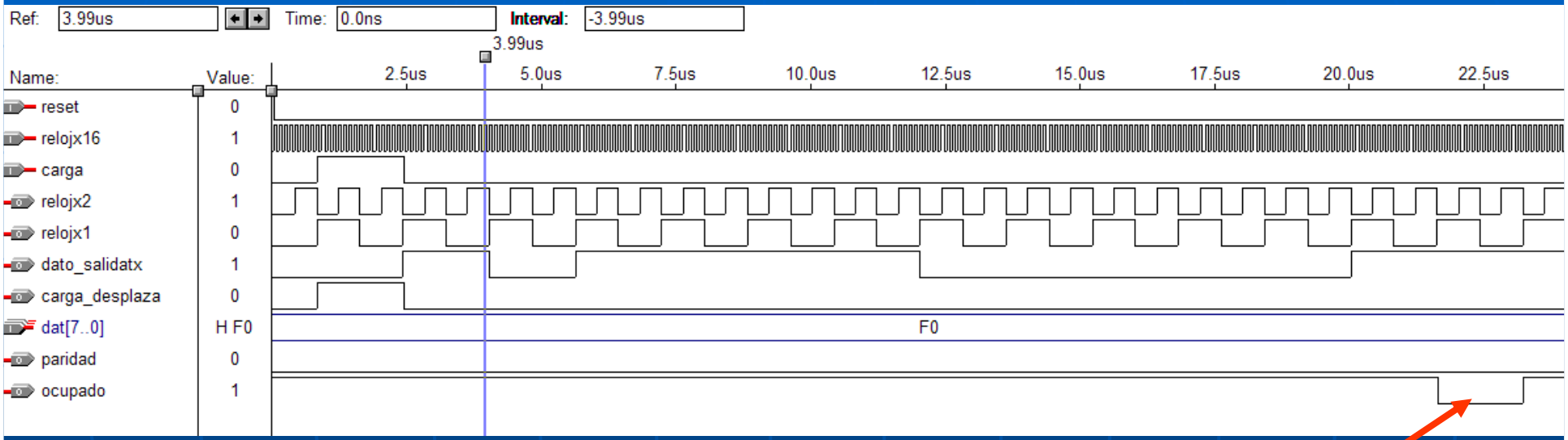
dato "F1"

P = "1"

En esta simulación de transmiten dos datos seguidos: "F0" y luego "F1".
El primero genera un bit P="0" y el segundo un bit P="1".

SIMULACIÓN DE TRANSMISOR DE LA UART

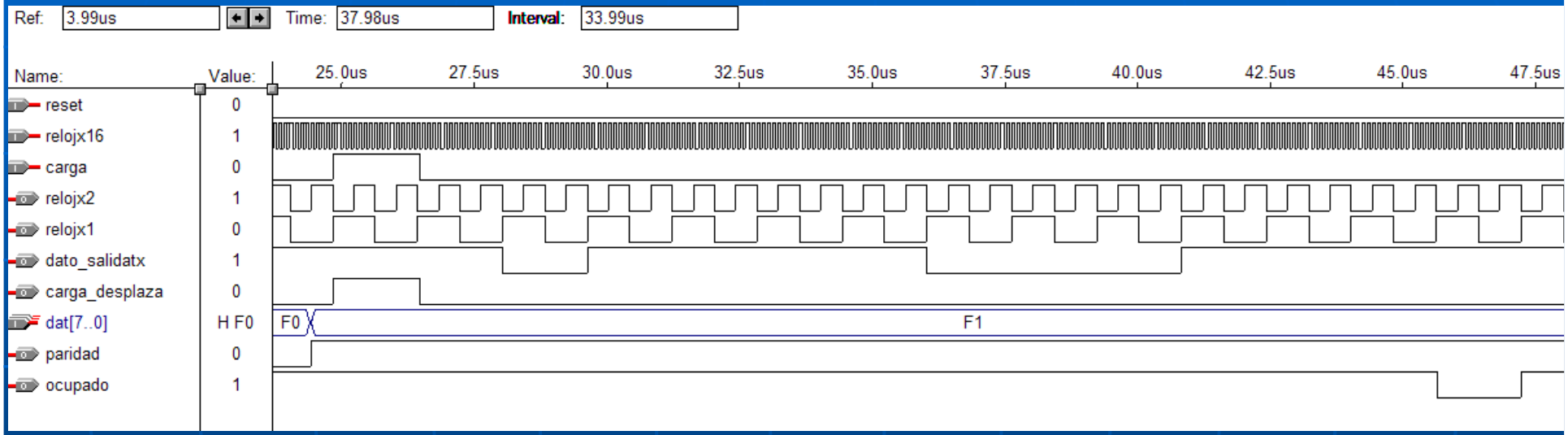
UART



Aquí avisa que terminó de transmitir

SIMULACIÓN DE TRANSMISOR DE LA UART

UART



```
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_shiftreg.inc";
INCLUDE "lpm_latch.inc";        % gate=1 habilita, gate=0 latch %

SUBDESIGN uart_02
(
    reset, relojx16, datx[7..0], carga                          : INPUT;
    relojx2, relojx1, dato_salidatx, carga_desplaza             : OUTPUT;
    paridad, ocupado                                            : OUTPUT;
    darx[7..0], pulsorx, zrx, habilita_rx, listo                : OUTPUT;
)
VARIABLE
    prescaler: lpm_counter WITH ( LPM_WIDTH=4);
    contadortx: lpm_counter WITH ( LPM_WIDTH=4);
    contadorrx: lpm_counter WITH ( LPM_WIDTH=4);
    contador_bits:  lpm_counter WITH ( LPM_WIDTH=4);
    desplazatx: lpm_shiftreg WITH ( LPM_WIDTH=12);
    desplazarx: lpm_shiftreg WITH ( LPM_WIDTH=12);
    latchx8: lpm_latch WITH ( LPM_WIDTH=8);
    habilita            : NODE;
    entrada_rx, yrx, yrxa, rst_rx, salida_rx[7..0]              : NODE;
    detecta                                                     : DFFE;

                %   estado   salida %
                %   actual   actual %
 ssrx: MACHINE OF BITS (zrx)
       WITH STATES (s0    =   0,
                    s1    =   0,
                    s2    =   1,
                    s3    =   0);

                %   estado   salida %
                %   actual   actual %
 ssrxa: MACHINE OF BITS (zrxa)
       WITH STATES (s0a    =   0,
                    s1a    =   0,
                    s2a    =   1,
                    s3a    =   0);
```

UART

DISEÑO DEL TRANSMISOR Y RECEPTOR

NOTA: El diseño se realizó considerando que los dos circuitos tienen el mismo reloj de referencia "relojx16" (funcionando como un loop-back).

UART

DISEÑO DEL TRANSMISOR Y RECEPTOR

```
BEGIN
    paridad = datx[0] $ datx[1] $ datx[2] $ datx[3] $ datx[4] $ datx[5] $ datx[6] $ datx[7];
    %Función Or-Exclusiva de 8 variables%

    desplazatx.clock = relojx1;
    desplazatx.data[11] = VCC;
    desplazatx.data[10] = GND;
    desplazatx.data[9..2] = datx[7..0];
    desplazatx.data[1] = paridad;
    desplazatx.data[0] = VCC;
    desplazatx.enable = habilita;
    desplazatx.load = carga_desplaza;
    prescaler.aclr = reset;
    prescaler.clock = relojx16;
    relojx2 = prescaler.q[2];
    relojx1 = prescaler.q[3];

    contadortx.aclr = reset;
    contadortx.clock =  relojx1;
    dato_salidatx = desplazatx.shiftout;

    IF carga == VCC THEN
        carga_desplaza = VCC;
        contadortx.aclr = VCC;
    ELSE
        carga_desplaza = GND;
    END IF;

    IF contadortx.eq[12] == VCC THEN
        habilita = GND;
    ELSE
        habilita = VCC;
    END IF;

    ocupado = habilita;
```

```
%--- Etapa de recepción --%

    entrada_rx = dato_salidatx;
    yrx = entrada_rx;

   TABLE
   %  estado  entrada      estado%
   %  actual  actual       próximo%
      ssrx,      yrx      =>   ssrx;
      s0,       0       =>   s0;
      s0,       1       =>   s1;
      s1,       0       =>   s2;
      s1,       1       =>   s1;
      s2,       0       =>   s3;
      s2,       1       =>   s3;
      s3,       0       =>   s3;
      s3,       1       =>   s0;
   END TABLE;

   TABLE
   %  estado  entrada      estado%
   %  actual  actual       próximo%
      ssrxa,      yrxa       =>   ssrxa;
      s0a,       0       =>   s0a;
      s0a,       1       =>   s1a;
      s1a,       0       =>   s2a;
      s1a,       1       =>   s1a;
      s2a,       0       =>   s3a;
      s2a,       1       =>   s3a;
      s3a,       0       =>   s3a;
      s3a,       1       =>   s0a;
   END TABLE;

    contadorrx.clock = relojx16;
    contadorrx.aclr = reset;

    ssrx.clk = relojx16;
    ssrx.reset = reset;
```

UART

DISEÑO DEL TRANSMISOR Y RECEPTOR

La máquina de estados “ssrx” genera un pulso cada vez que se detecta una transición 1 ➔ 0 de la entrada al receptor.

“ssrxa” detecta cuando baja la línea de “habilita_rx” lo que indica que terminó el RD de cargar la información.
De este modo su salida “zrxa” genera un pulso para avisar al microprocesador que está listo para ser leído.

```
    ssrxa.clk = relojx16;
    ssrxa.reset = reset;

    yrxa = habilita_rx;
    listo = zrxa;

    detecta.clrn = !(reset # rst_rx);
    detecta.d = VCC;
    detecta.clk = zrx;
    habilita_rx = detecta.q;

    IF habilita_rx == VCC & contadorrx.eq[8] == 1 THEN
        pulsorx = VCC;
    ELSE pulsorx = GND;
    END IF;

    IF contador_bits.q[] == 11 THEN
        rst_rx = VCC;
    ELSE
        rst_rx = GND;
    END IF;

    desplazarx.clock = pulsorx;
    desplazarx.shiftin = entrada_rx;

    salida_rx[7..0] = desplazarx.q[9..2];

    contador_bits.aclr = reset;
    contador_bits.clock = pulsorx;

    latchx8.data[] = salida_rx[];
    darx[] = latchx8.q[];
    latchx8.gate = listo;
END;
```

UART

DISEÑO DEL TRANSMISOR Y RECEPTOR

Al detectar el posible bit de Start comienza a generarse un reloj de comando para el RD “desplazarx”. Este reloj “pulsorx” genera un pulso en base a un “relojx16” que es 16 veces mas rápido que la cadencia de datos de entrada.

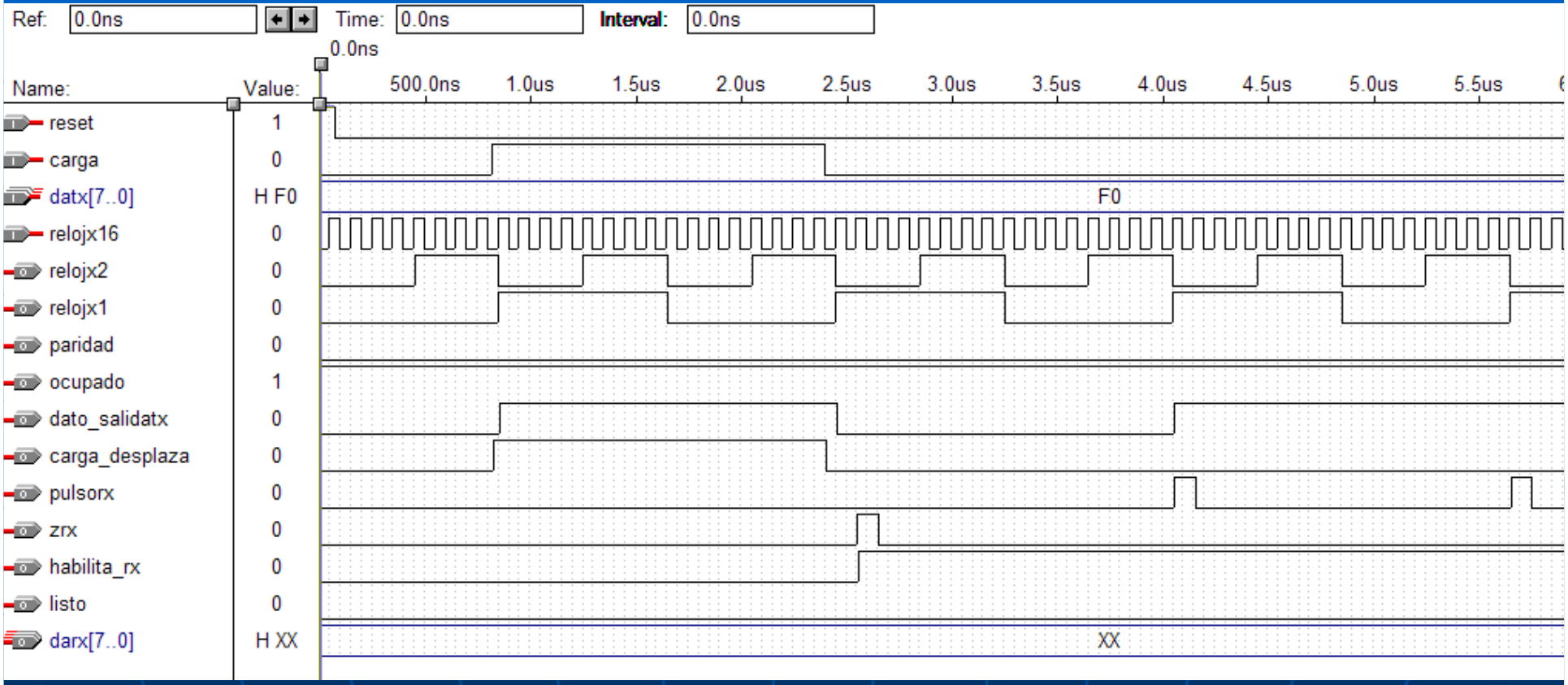
A fin de muestrear en la mitad del “tiempo de bit” “pulsorx” genera un pulso cada tiempo de bit, ubicado en la mitad del mismo con una duración de un ciclo de “relojx16”.

SIMULACIÓN DE TRANSMISOR Y RECEPTOR DE LA UART

UART

SIMULACIÓN DE TRANSMISOR Y RECEPTOR DE LA UART

UART



UART

DISCUSIÓN:

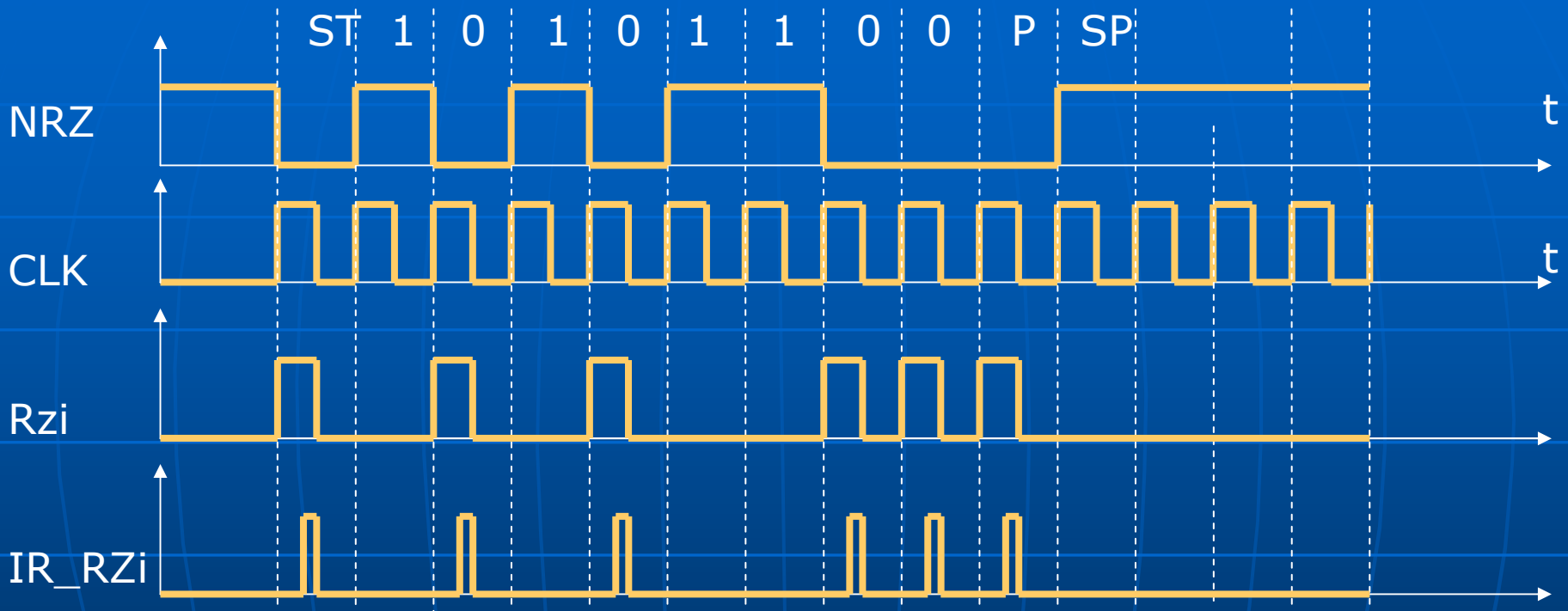
Cómo se podría usar la UART para una comunicación con enlace infrarrojo donde hay que preservar el consumo de potencia en el TX???

Posible solución:

√ Empleo de formato RZi para minimizar potencia.

√ Transmisión de pulso para el cero de 3/16 del tiempo de bit.

UART infrarroja



Un "0" se lo codifica con un pulso positivo en medio del tiempo de bit.
Un "1" se lo codifica con un nivel nulo de señal.

PORQUË??: Porque en RS232 un "1" está siempre generándose cuando no hay actividad en el enlace ➔ conviene no gastar potencia…!!!!

UART infrarroja

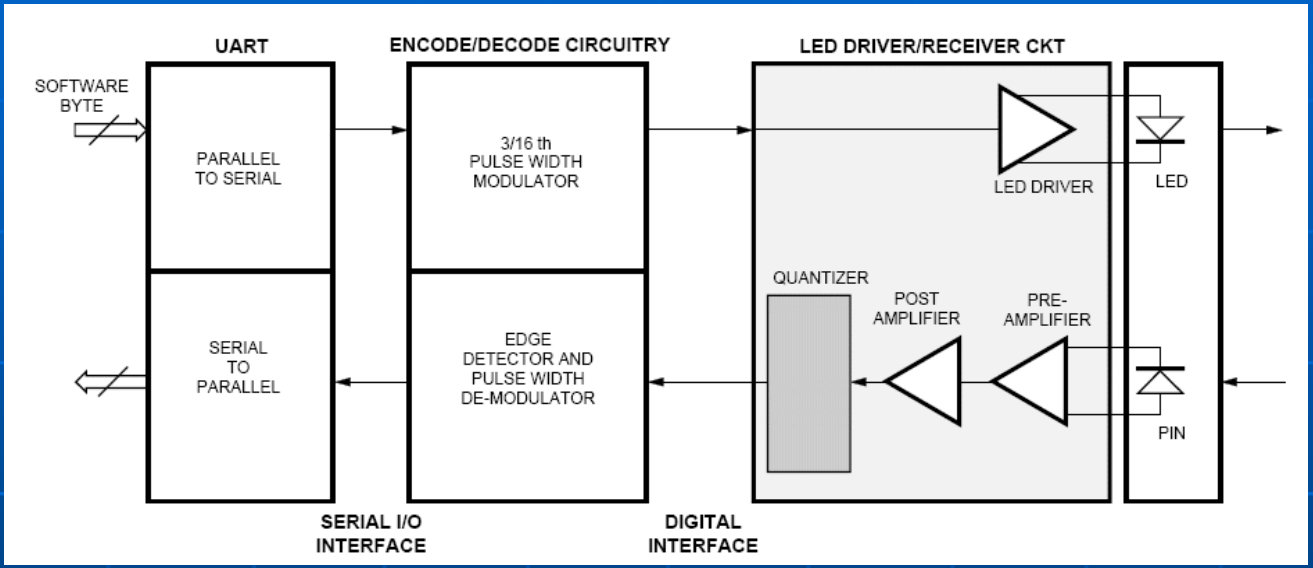
Con este esquema es posible aumentar la potencia pico del TX manteniendo el consumo promedio.

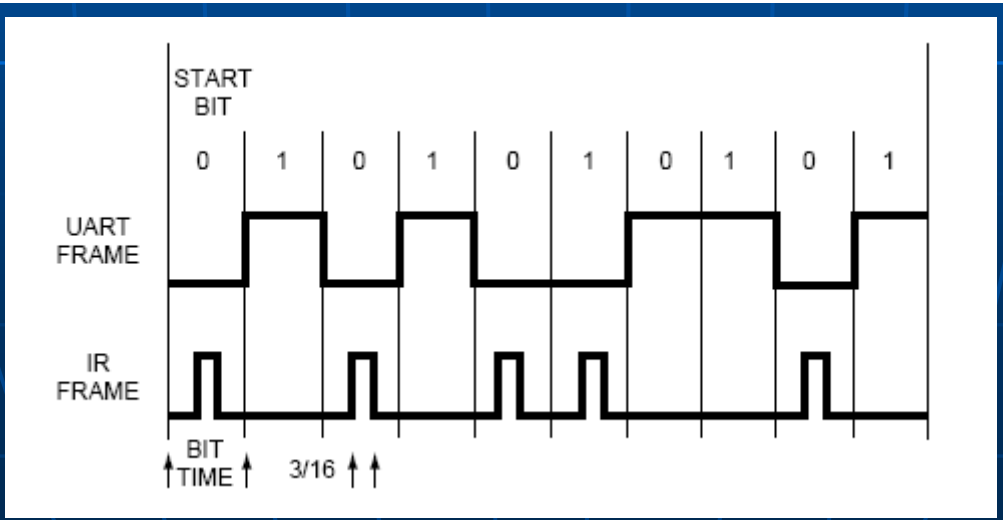
El diseño del TX se puede modificar partiendo de la salida del RD "desplazatx".
Con esta señal y relojx16 se pueden generar los pulsos positivos de por ejemplo 3 ciclos de relojx16 de duración cada vez que haya un "0" para enviar.

El diseño del RX debe partir en procesar la información desde el momento en que se detecta una transición de "0" a "1" ó viceversa dependiendo de cómo se obtenga el dato desde el fotoreceptor.

UART infrarroja

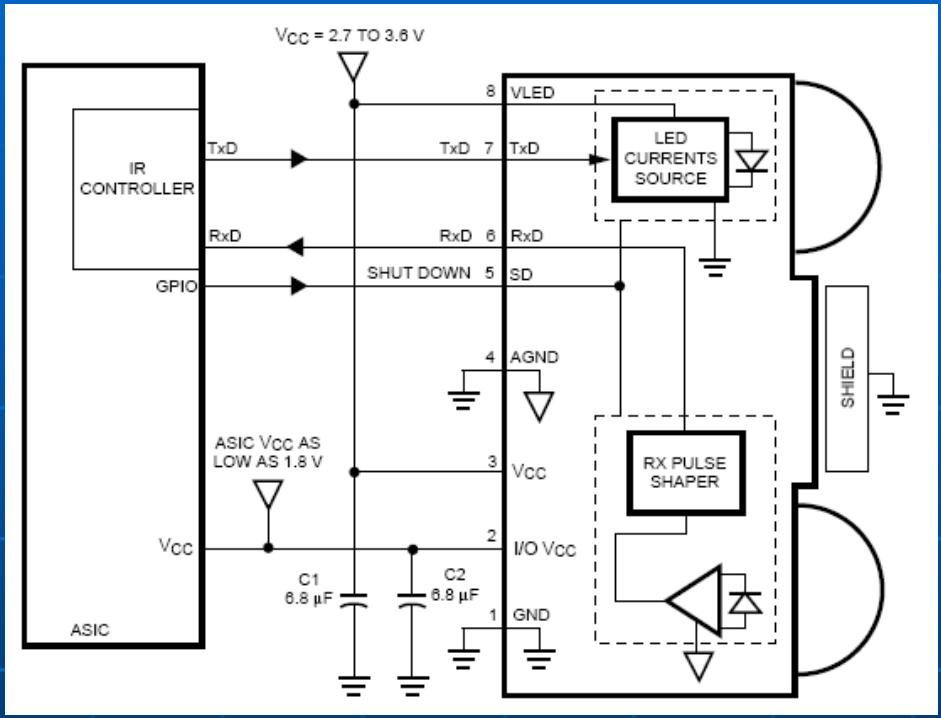
Ejemplo IrDA Data Link de Agilent

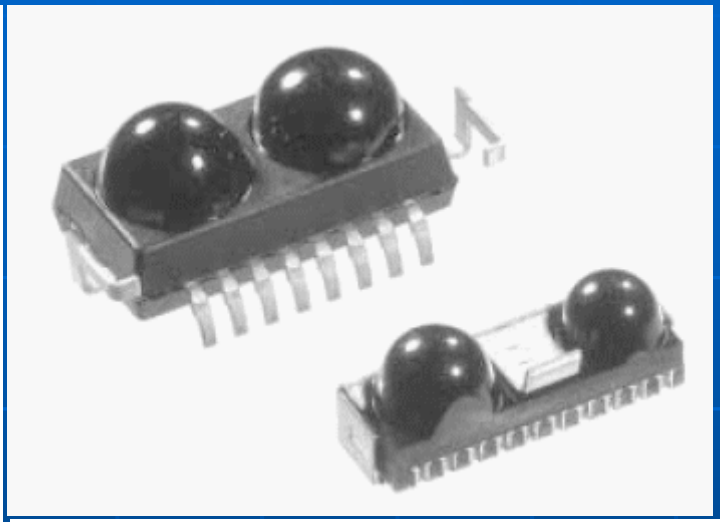


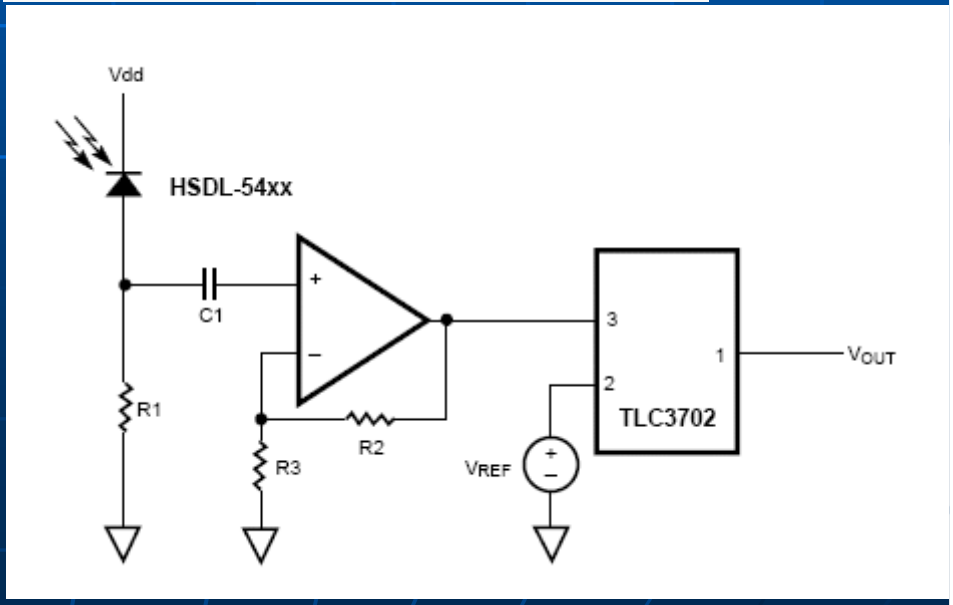


UART infrarroja

Ejemplo IrDA Data Link de Agilent







```
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_shiftreg.inc";
INCLUDE "lpm_latch.inc";        % gate=1 habilita, gate=0 latch %

SUBDESIGN uart_03
(
    reset, relojx16, dat[7..0], carga                          : INPUT;
    relojx2, relojx1, dato_salidatx, carga_desplaza            : OUTPUT;
    paridad, ocupado, salida0_rzi, habilita, salida_txrzi      : OUTPUT;
)
VARIABLE
    prescaler: lpm_counter WITH ( LPM_WIDTH=4);
    contadortx: lpm_counter WITH ( LPM_WIDTH=4);
    contador_rzi: lpm_counter WITH ( LPM_WIDTH=4);
    desplaza: lpm_shiftreg WITH ( LPM_WIDTH=12);
    detecta                                    : DFFE;

BEGIN
    paridad = dat[0] $ dat[1] $ dat[2] $ dat[3] $ dat[4] $ dat[5] $ dat[6] $ dat[7];
    %Función Or-Exclusiva de 8 variables%

    desplaza.clock = relojx1;
    desplaza.data[11] = VCC;
    desplaza.data[10] = GND;
    desplaza.data[9..2] = dat[7..0];
    desplaza.data[1] = paridad;
    desplaza.data[0] = VCC;
    desplaza.enable = habilita;
    desplaza.load = carga_desplaza;
    prescaler.aclr = reset;
    prescaler.clock = relojx16;
    relojx2 = prescaler.q[2];
    relojx1 = prescaler.q[3];
```

UART infrarroja (posible solución para TX)

Salida al driver infrarrojo

Sirve para la generación de Rzi al TX infrarrojo

UART infrarroja (posible solución para TX)

```
contadortx.aclr = reset;
contadortx.clock =  relojx1;
dato_salidatx = desplaza.shiftout;

contador_rzi.clock = relojx16;

IF carga == VCC THEN
    carga_desplaza = VCC;
    contadortx.aclr = VCC;
ELSE
    carga_desplaza = GND;
END IF;

IF contadortx.eq[12] == VCC THEN
    habilita = GND;
ELSE
    habilita = VCC;
END IF;

ocupado = habilita;
```

Esta sección es idéntica a la anterior.

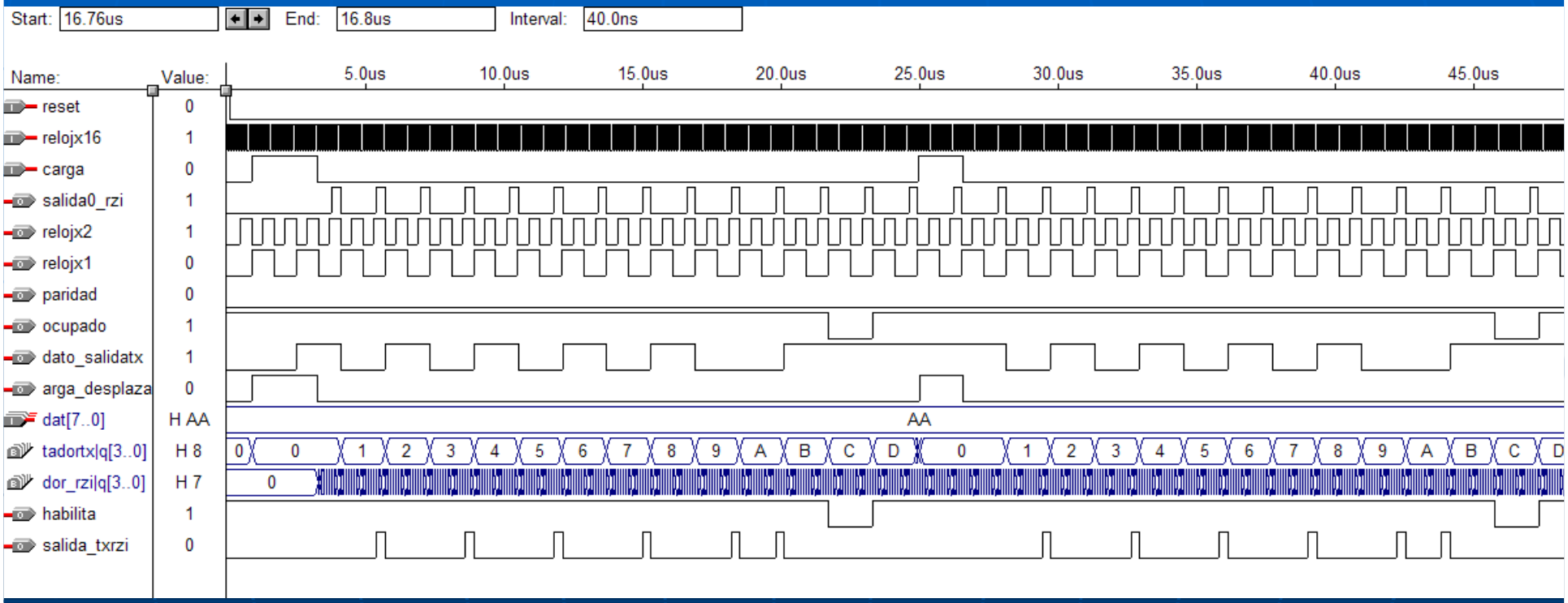
UART infrarroja (posible solución para TX)

```
%  Sección para conversión RZi con
   modulación 3/16                  %

    detecta.clk = !carga;
    detecta.clrn = !reset;
    detecta.d = VCC;

    contador_rzi.aclr = !detecta.q;

    IF (contador_rzi.q[] > H"5" & contador_rzi.q[] < H"9") THEN
            salida0_rzi = VCC;
    ELSE
            salida0_rzi = GND;
    END IF;

    CASE dato_salidatx IS
        WHEN B"0" =>
            salida_txrzi = salida0_rzi;
        WHEN B"1" =>
            salida_txrzi = GND;
    END CASE;
END;
```

Con CASE se conmuta la salida a un "0" ó a un "pulso" dependiendo si la salida del RD es "1" ó "0" respectivamente
El pulso es generado con el "contador_rzi" de 4 bits tal que cuando su conteo sea "6", "7" u "8" ponga un "1" en la señal "salida0_rzi" caso contrario, en los otros 15 valores posibles será "0" (código 3/16).
Dicho contador se lo habilita recién cuando se recibe la órden de "carga" para evitar que se generen pulsos sin que el RD esté activo todavía. Esto se logra con el FFD "detecta". La sincronización del pulso generado en "salida_rzi" se obtiene usando como reloj de "detecta" a la señal de "carga".
Cuando "carga" baja, la salida de "detecta" se pone en "1" y deja de forzar a "contador_rzi" a un reset ya que este es un reset asincrónico.

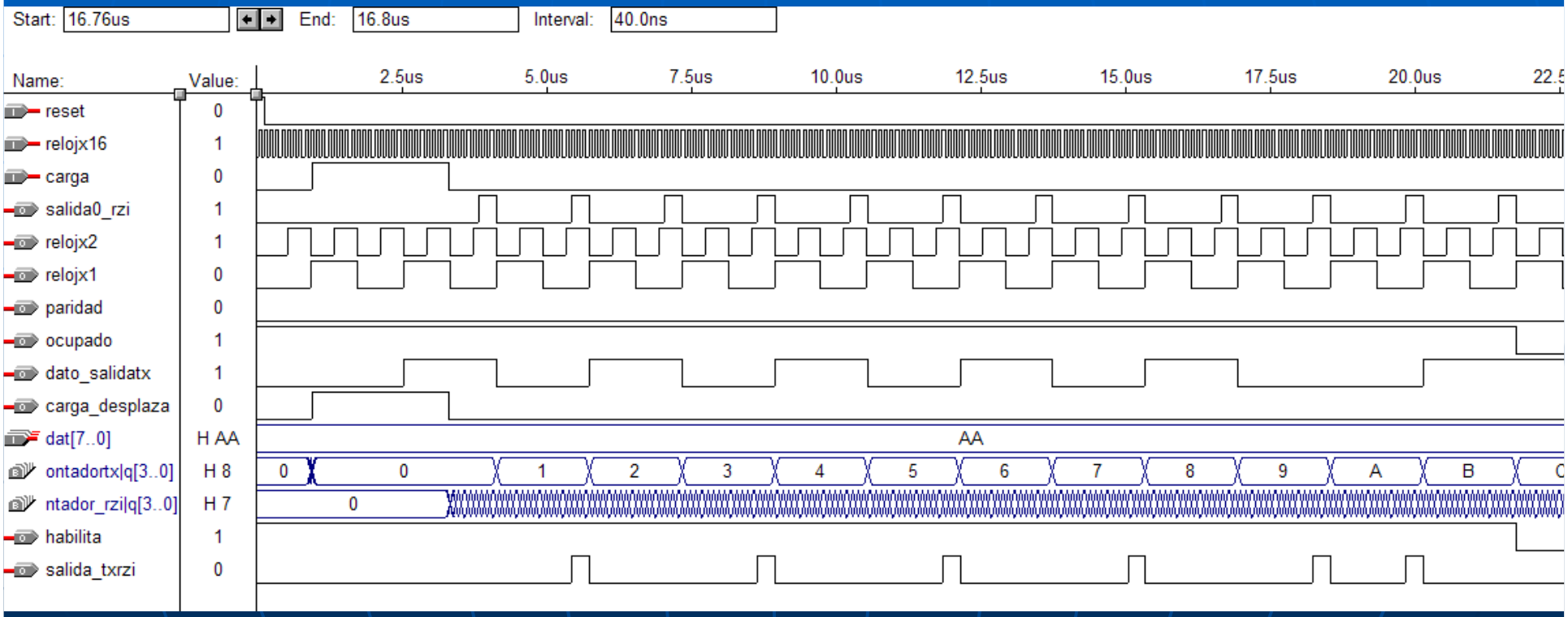
SIMULACIÓN DE TRANSMISOR DE LA UART CON SALIDA PARA INFRARROJO

UART infrarroja (posible solución para TX)



SIMULACIÓN DE TRANSMISOR DE LA UART CON SALIDA PARA INFRARROJO

UART infrarroja (posible solución para TX)



Aplicaciones con Lógica Programable en Comunicaciones

CONCLUSIONES:

- √ Se ha presentado posibles soluciones para la implementación de un modulador-demodulador PCM, un codec Manchester y un transceiver UART elemental, los cuales transmiten señales digitales en formato serie.
  En los 3 casos las interfaces de entrada-salida son en formato paralelo para poder ser conectadas a microprocesadores.

- √ Se ha empleado para ello el lenguaje de descripción de hardware de Altera AHDL a fin de mostrar como es posible diseñar a través del uso de un lenguaje textual.

NOTA IMPORTANTE:
El propósito de este seminario no es el de mostrar reglas de diseño optimizadas.
Ex profeso se han utilizado varios tipos de componentes de HDL para incluírlos en forma didáctica.

BIBLIOGRAFÏA:

Nota de aplicación: “XAPP341” de Xilinx.
Nota de aplicación: “XAPP345” de Xilinx.
Nota de aplicación: “5988-9321EN” de Agilent.
Guía de Help de Max-PlussII de Altera.
Apuntes de clases de laboratorio: Cátedra ISLD.
Trabajo final “Introducción a dispositivos FPGA”. Cátedra ISLD.