



# **Comunicazione Seriale VHDL - UART**

Giovanni De Luca



# Storia dello standard RS-232

- L'interfaccia seriale EIA RS-232 è uno standard costituito da una serie di protocolli meccanici, elettrici ed informatici che rendono possibile lo scambio di informazioni a bassa velocità tra dispositivi digitali.
- Nel corso di oltre 40 anni lo standard si è evoluto, pur mantenendosi in larga parte invariato. L'evoluzione è riconoscibile dalla sigla, leggendo l'ultima lettera; l'ultima revisione è del 1997 ed è indicata come EIA RS-232f. Probabilmente la versione più diffusa è la RS232c, del 1969, che corrisponde alle specifiche europee CCITT, (**Consultative Committee for International Telephony and Telegraphy**) raccomandazione V.24.
- Pur essendo un protocollo piuttosto vecchio, attualmente la EIA RS-232 è ancora largamente utilizzata per la comunicazione a bassa velocità tra microcontrollori, dispositivi industriali ed altri circuiti relativamente semplici, che non necessitano di particolare velocità;



# Descrizione

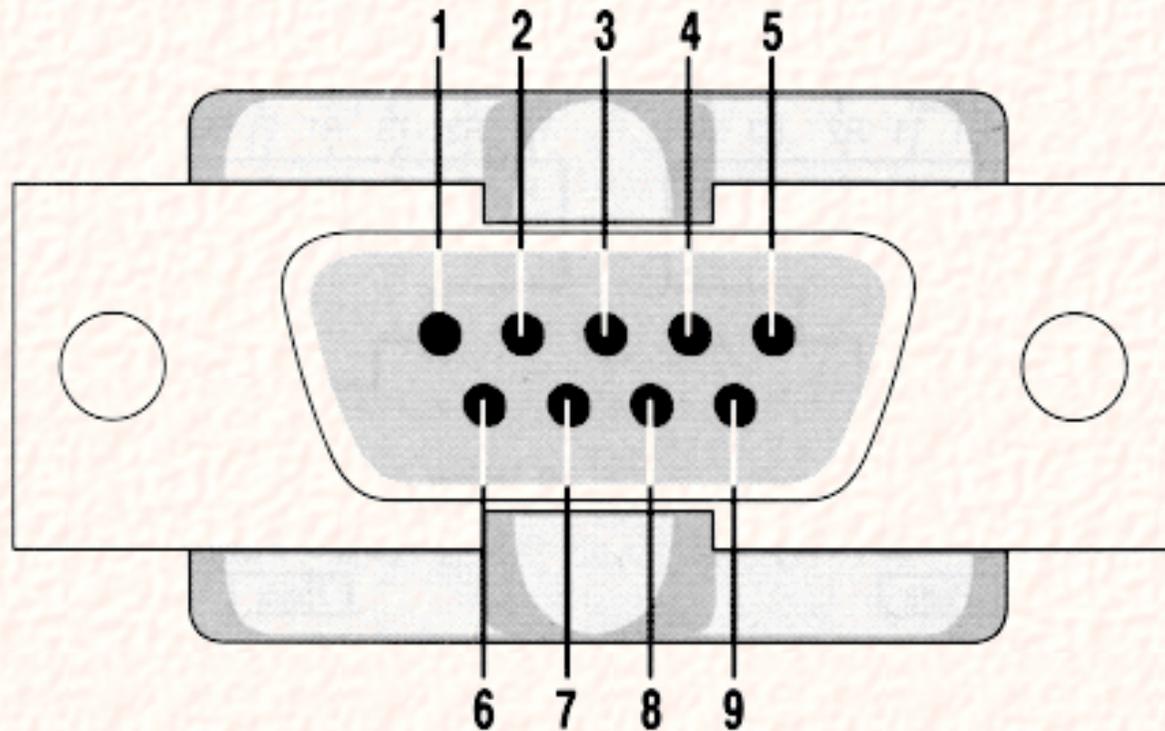
- L'interfaccia EIA RS-232 ridotta (solo asincrona) utilizza un protocollo di trasmissione seriale di tipo asincrono.
- Seriale significa che i bit che costituiscono l'informazione sono trasmessi uno alla volta su di un solo "filo". Questo termine è in genere contrapposto a "parallelo": in questo caso i dati sono trasmessi contemporaneamente su più fili, per esempio 8, 16 o 32.
- Si potrebbe pensare che la trasmissione seriale sia intrinsecamente più lenta di quella parallela (su di un filo possono passare meno informazioni che su 16). In realtà questo non è vero in assoluto, soprattutto a causa della difficoltà di controllare lo skew (disallineamento temporale tra i vari segnali) dei molti trasmettitori in un bus parallelo, e dipende dalle tecnologie adottate
- Asincrono significa, che i dati sono trasmessi, byte per byte, in modo anche non consecutivo e senza l'aggiunta di un segnale di clock, cioè di un segnale comune che permette di sincronizzare la trasmissione con la ricezione; ovviamente sia il trasmettitore che il ricevitore devono comunque essere dotati di un clock locale per poter interpretare i dati. La sincronizzazione dei due clock è necessaria ed è fatta in corrispondenza della prima transizione sulla linea dei dati.



# Unità di misura

- Le unità di misura della velocità di trasmissione sono essenzialmente due: il **baud** ed il **bit per secondo** (bps o, più raramente, b/s), spesso trattate erroneamente come sinonimi.
- Il **baud rate** indica il numero di caratteri, simboli, numeri trasmessi al secondo; il **bps**, o **bit rate**, indica, come dice il nome, quanti bit al secondo sono trasmessi lungo la linea. I due termini non sono equivalenti, in quanto il baud rate è riferito al contenuto del messaggio, mentre il bit rate è riferito alla struttura logico-temporale che ne permette la trasmissione.
- Ad esempio, nel caso di una trasmissione TTY (telescrivente) che utilizza il codice **Baudot** (5 bit), per ogni carattere trasmesso, sono necessari: 1 bit di start, 5 bit di dati (il carattere da inviare), 1,5 bit di stop. Totale 7,5 bit. Se si utilizza, invece un codice a 8 bit (es. ASCII), occorrono: 1 bit di start, 8 bit di dati (il carattere da inviare), 1 bit di stop. Totale 10 bit.

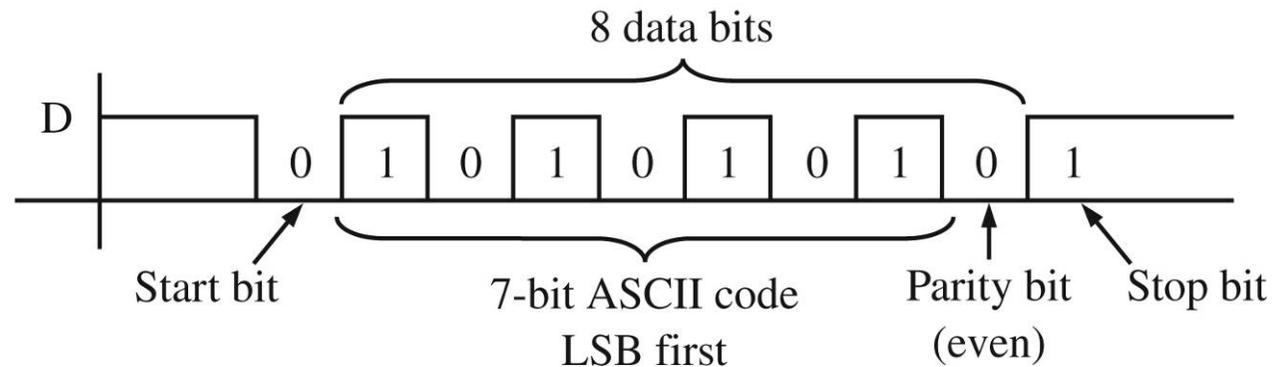
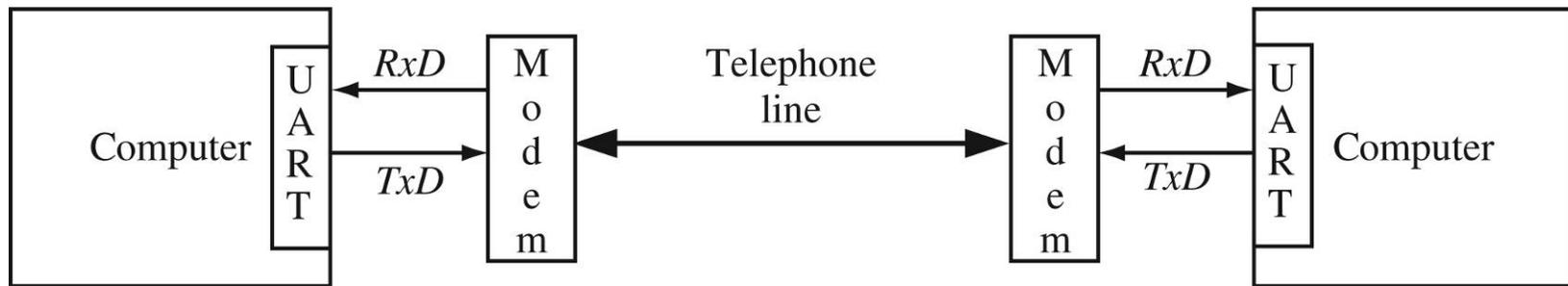
# RS-232



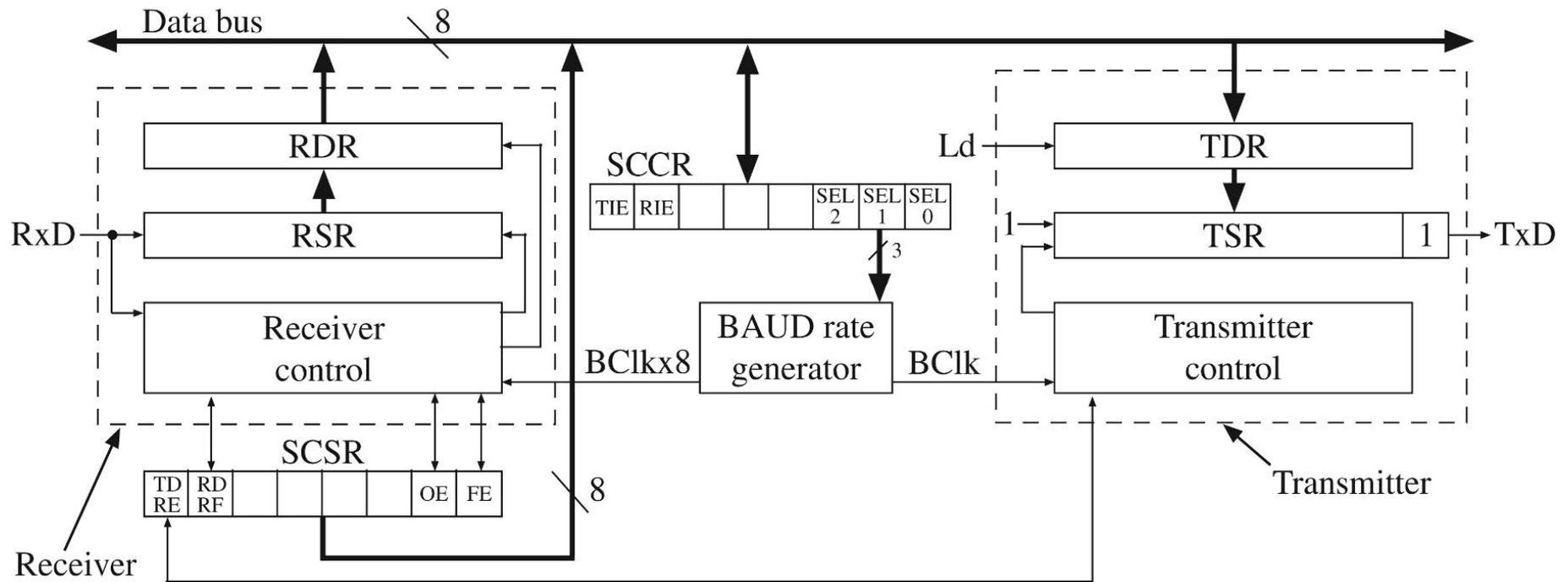
Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

# UART

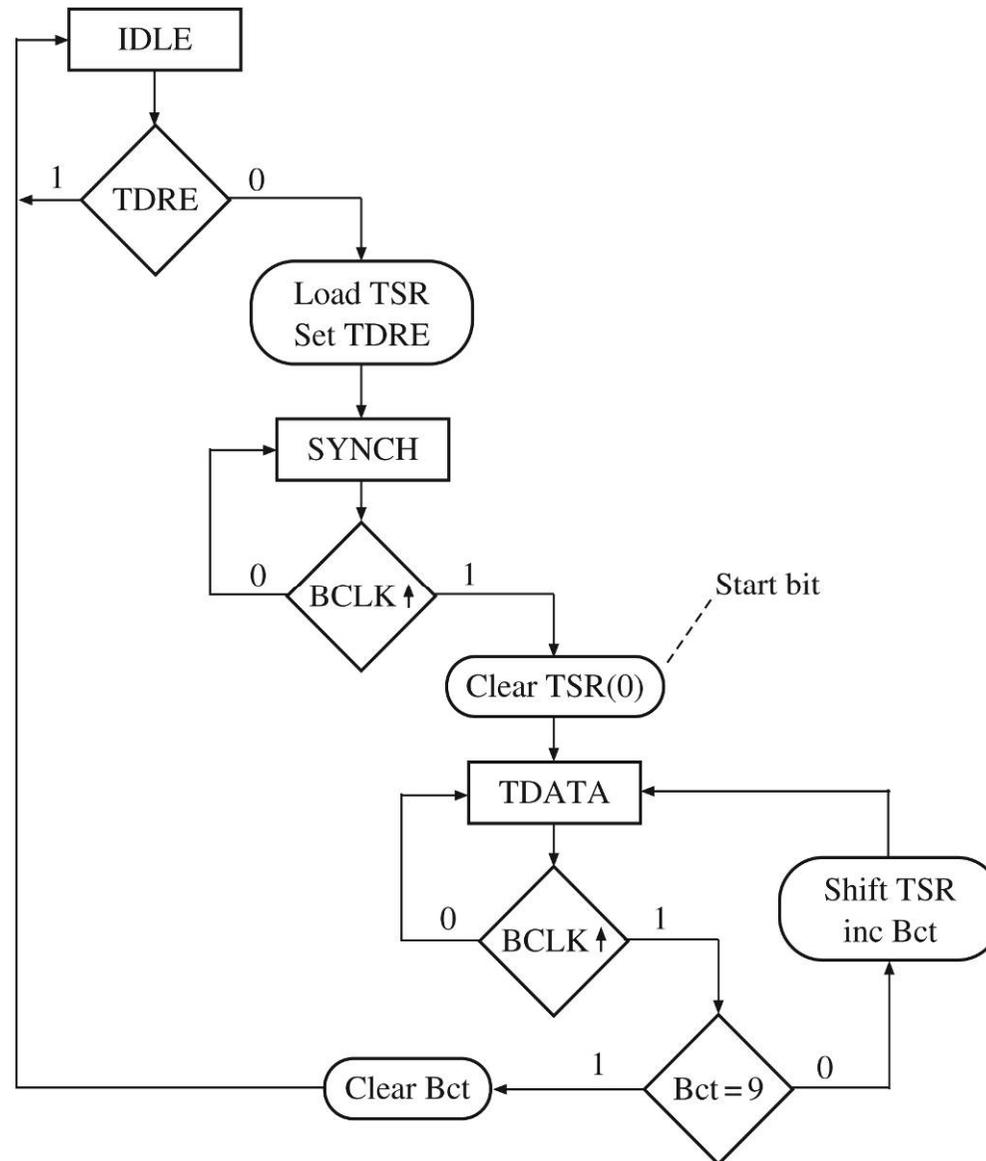
- Universal Asynchronous Receiver Transmitter
  - Serial Data Transmission



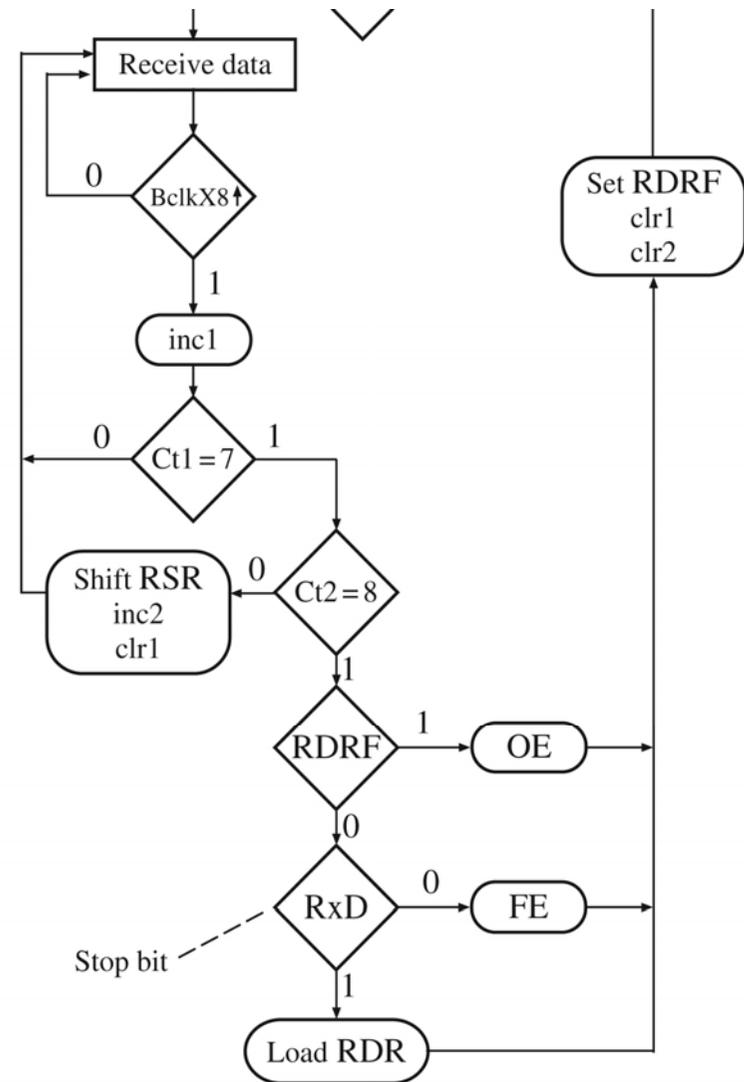
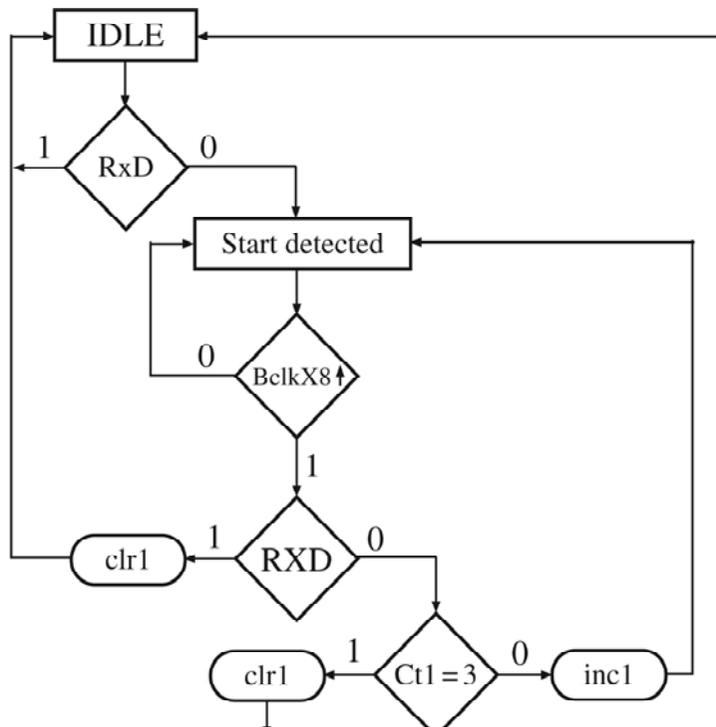
# UART Block Diagram



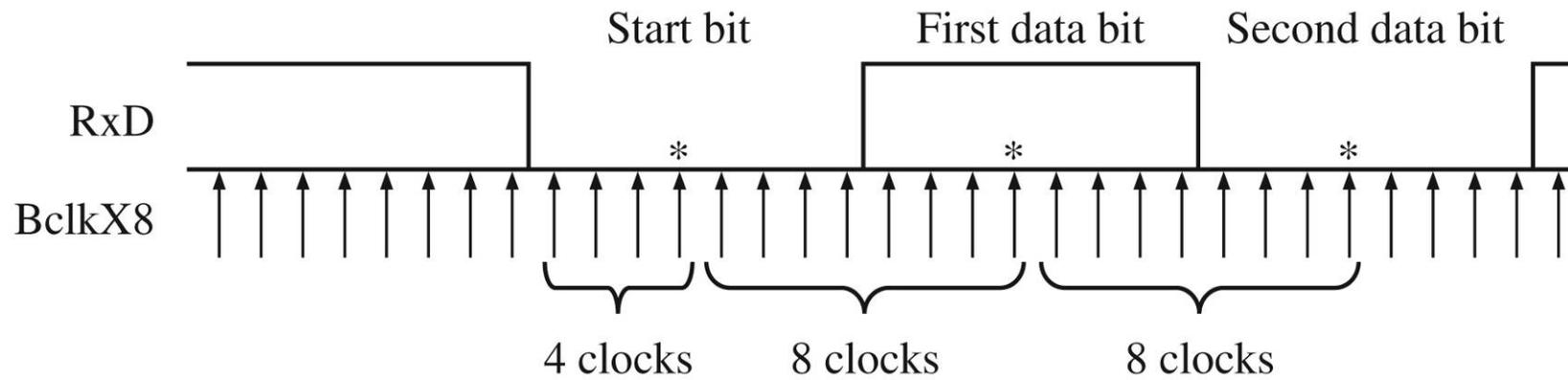
# Transmitter SM Chart



# Receiver SM Chart



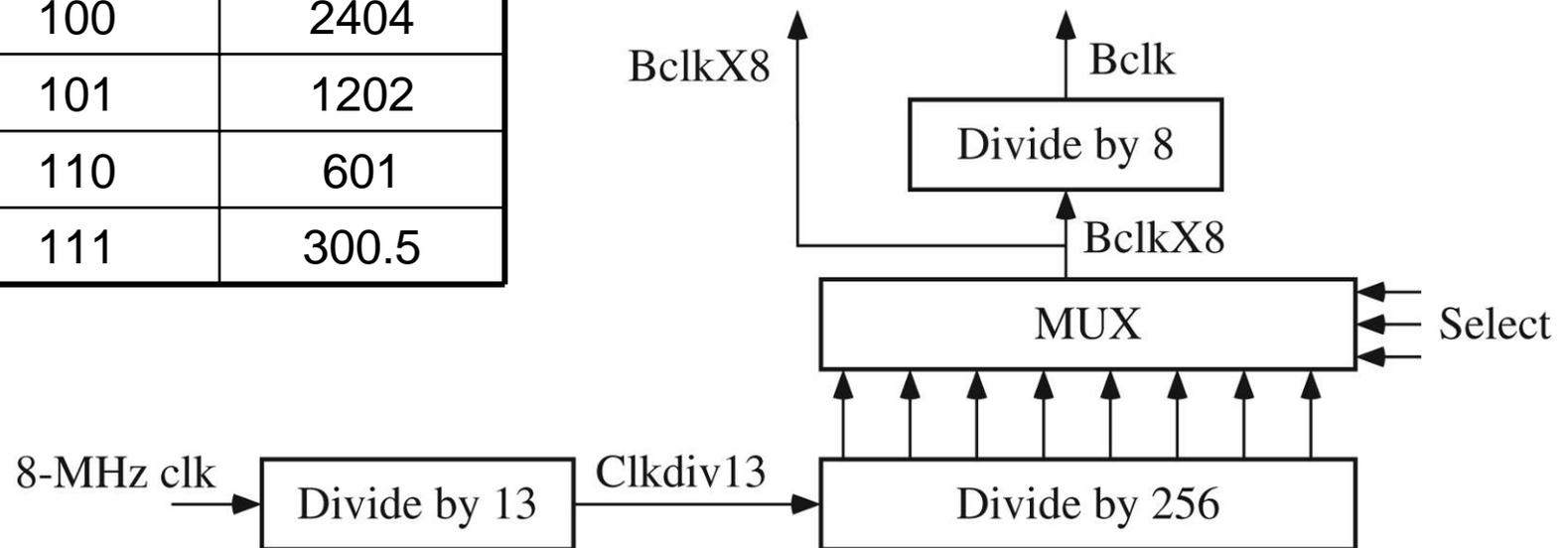
# Sampling RxD



\*Read data at these points

# Baud Rate Generator

Select Bits	BAUD Rate
000	38,462
001	19,231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5





# UART/USART

- Standard Universal Synchronous/Asynchronous Receiver/Trasmitter (USART)
- Progettata per le comunicazioni con le famiglie di microprocessori INTEL
- Device Periferico programmabile da CPU



# UART/USART Trasmissione

- USART accetta dati dalla CPU in formato PARALLELO e
- li converte in uno stream di dati seriali



## **UART/USART Ricezione**

- USART riceve un flusso di dati seriali e li
- converte in formato parallelo per inviarli alla CPU



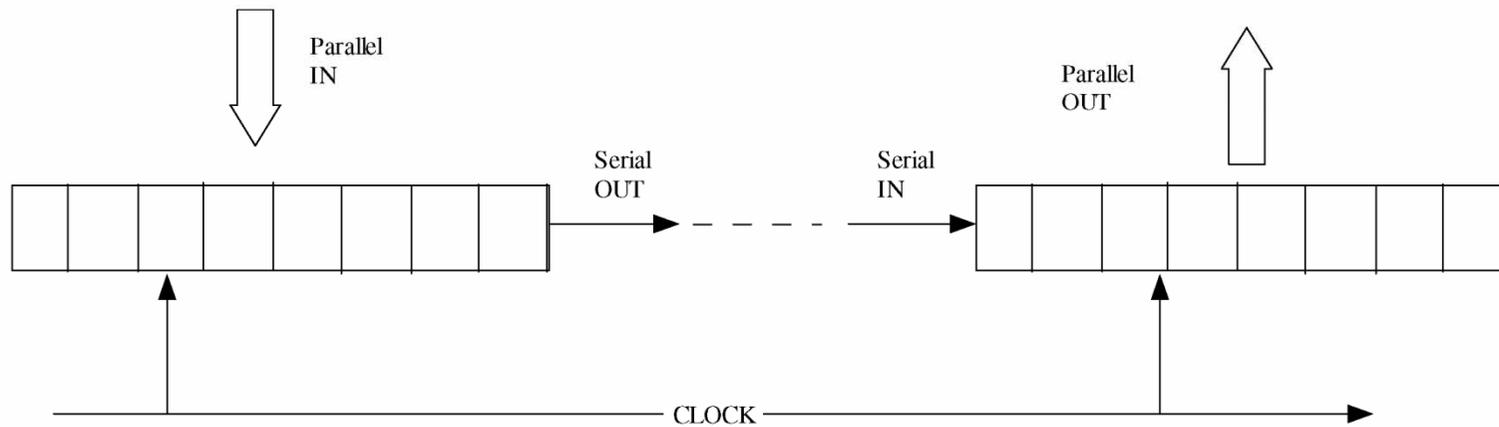
# UART/USART

- USART segnala alla CPU se può accettare un nuovo carattere da trasmettere, oppure
- Se ha ricevuto un nuovo carattere che può comunicare alla CPU
- La CPU può leggere lo stato completo della USART in ogni istante
  - Data Transmission Error
  - Segnali di Controllo



## Modalità di trasmissione

- Simplex: esiste una linea monodirezionale (1 trasmette e 1 riceve)
- Half Duplex: esiste una linea bidirezionale (1 trasmette e 1 riceve ma possono scambiarsi i ruoli)
- Full Duplex: esiste una linea bidirezionale entrambi possono trasmettere e ricevere contemporaneamente.

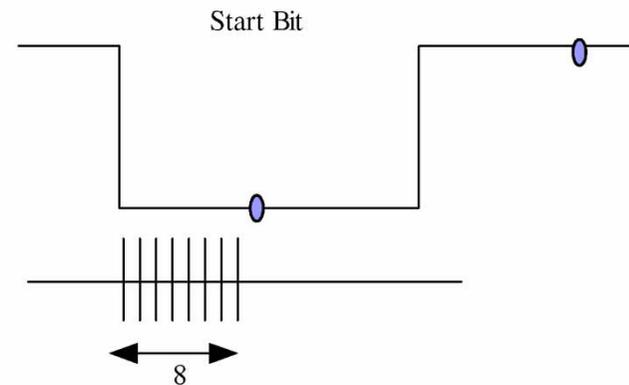


In realtà non si trasmette il clock ma il RX è dotato di oscillatori locali che generano un clock di campionamento adeguato oppure sono in grado di sincronizzarsi con il Trasmettitore.

L'informazione elementare che viene trasmessa è "un carattere" codificato su massimo 8 bit ( start, msg, controllo, coda )

# Comunicazione Asincrona

- Il clock del RX viene settato ad una velocità molto superiore del TX (16x o 64x) in modo da non perdere la transizione 1->0 dello start bit,
- Poi il clock del RX rifasa il campionamento con un shift di 8 impulsi posizionandosi al centro del bit e campiona ogni 16 impulsi





## Comunicazione Asincrona (2)

- Sono necessarie delle informazioni aggiuntive per controllare che non si sia persa la sincronizzazione:
  - 1 bit di parità per ogni carattere inviato,
  - Controllo di parità per l'intero messaggio,
  - Controlli di parità longitudinale, aggiungendo byte CHECK SUM



# Comunicazione Sincrona

- Per aumentare le prestazioni e diminuire gli sprechi occorre:
  - Eliminare start e stop bits,
  - Eliminare bit di parità,
  - Rilievo errori più sicuro
- Nella comunicazione sincrona mandiamo solo messaggi “significativi” come flusso continuo di bit,
- la sincronizzazione viene fatta solo all’inizio oppure dopo un numero **PREDETERMINATO** di caratteri inviando una sequenza di caratteri speciali (BYTE SYNC)
- Per la correzione degli errori: si applica un codice ridondante sull’intero frame (stesso numero di bit indipendentemente dalla lunghezza del msg).



# Tipologie di errori

- **OVERRUN**: il dato è stato copiato dallo shift register nel registro in prima che quest'ultimo fosse stato svuotato dalla CPU (dato perso!);
- **FRAMING**: si è persa la sincronizzazione;
- **PARITY**: è fallito il controllo sul bit di parità;

NOTA: tipicamente la comunicazione seriale avviene su canali molto rumorosi (linea telefonica).... Problema delle tensioni adeguate



# Architettura di una UART/USART

- 4 sezioni:
  - Scambio dati e comandi con il PC,
  - Trasmitter: contiene uno shift register ed un buffer temporaneo
  - Receiver: contiene uno shift register ed un buffer temporaneo
  - Rate generator: genera la frequenza di trasmissione

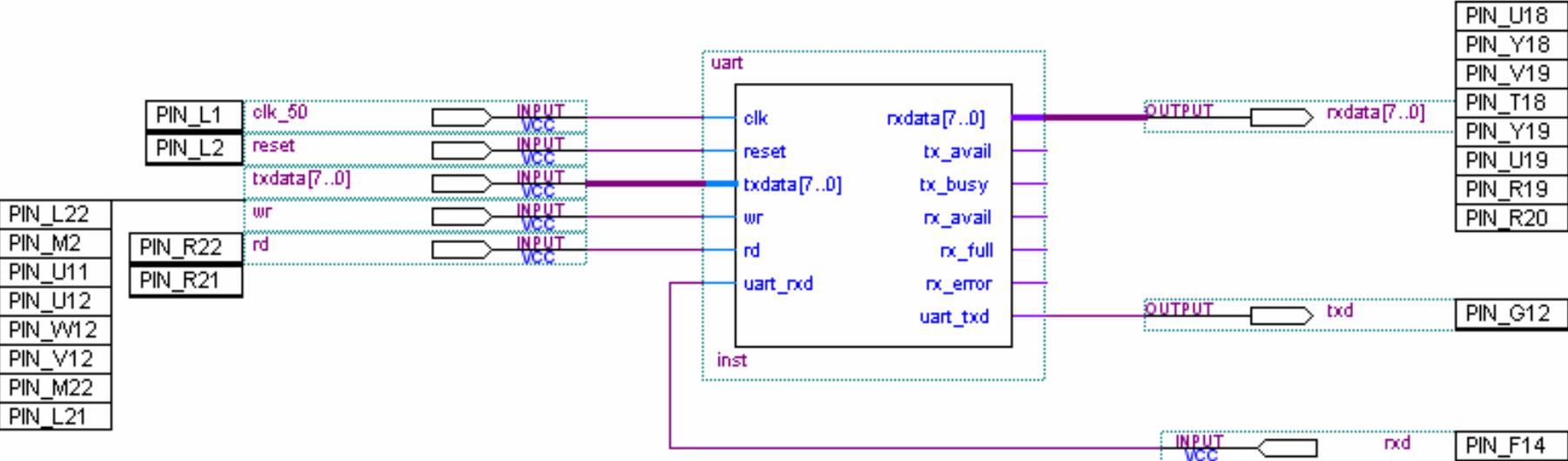


# Segnali di comunicazione

## RS-232

- TD ed RD per trasmettere e ricevere dati,
- RTS(Request to Send): indica intenzione a trasmettere;
- CTS(Clear to Send): indica lo stato di pronto a ricevere;
- DSR (Data set Ready): Tx ed RX sono collegati in trasmissione
- DTR (Data Terminal Ready): pronto a comunicare

# Quartus project di un semplice UART controller





## VHDL (uart.vhd)

```
-----  
library ieee;  
use ieee.std_logic_1164.all;  
  
-----  
-- UART  
entity uart is  
    generic (  
        divisor    : integer := 434 );  
    port (  
        clk        : in  std_logic;  
        reset      : in  std_logic;  
        txdata     : in  std_logic_vector(7 downto 0);  
        rxdata     : out std_logic_vector(7 downto 0);  
        wr         : in  std_logic;  
        rd         : in  std_logic;  
        tx_avail   : out std_logic;  
        tx_busy    : out std_logic;  
        rx_avail   : out std_logic;  
        rx_full    : out std_logic;  
        rx_error   : out std_logic;  
        uart_rxd   : in  std_logic;  
        uart_txd   : out std_logic );  
  
end uart;
```



**architecture rtl of uart is**

```
-----  
-- component declarations -----  
component uart_rx is  
    generic (  
        fullbit    : integer );  
    port (  
        clk        : in  std_logic;  
        reset      : in  std_logic;  
        dout       : out std_logic_vector(7 downto 0);  
        avail      : out std_logic;  
        error      : out std_logic;  
        clear      : in  std_logic;  
        rxd        : in  std_logic );  
end component;  
-----  
component uart_tx is  
    generic (  
        fullbit    : integer );  
    port (  
        clk        : in  std_logic;  
        reset      : in  std_logic;  
        din        : in  std_logic_vector(7 downto 0);  
        wr         : in  std_logic;  
        busy       : out std_logic;  
        txd        : out std_logic );  
end component;
```



-----  
*-- local signals* -----

**signal** utx\_busy : std\_logic;

**signal** utx\_wr : std\_logic;

**signal** urx\_dout : std\_logic\_vector(7 **downto** 0);

**signal** urx\_availl : std\_logic;

**signal** urx\_clear : std\_logic;

**signal** urx\_error : std\_logic;

**signal** txbuf : std\_logic\_vector(7 **downto** 0);

**signal** txbuf\_full : std\_logic;

**signal** rxbuf : std\_logic\_vector(7 **downto** 0);

**signal** rxbuf\_full : std\_logic;

**begin**

```

iotxproc: process(clk, reset) is
begin
    if reset='1' then
        utx_wr      <= '0';
        txbuf_full <= '0';

        urx_clear  <= '0';
        rxbuf_full <= '0';

    elsif clk'event and clk='1' then
        -- TX Buffer Logic
        if wr='1' then
            txbuf      <= txdata;
            txbuf_full <= '1';
        end if;
        if txbuf_full='1' and utx_busy='0' then
            utx_wr      <= '1';
            txbuf_full <= '0';
        else
            utx_wr      <= '0';
        end if;
        -- RX Buffer Logic
        if rd='1' then
            rxbuf_full <= '0';
        end if;
        if urx_avail='1' and rxbuf_full='0' then
            rxbuf      <= urx_dout;
            rxbuf_full <= '1';
            urx_clear  <= '1';
        else
            urx_clear <= '0';
        end if;
    end if;
end process;

```



```
uart_rx0: uart_rx
    generic map (
        fullbit => divisor )
    port map (
        clk      => clk,
        reset    => reset,
        dout     => urx_dout,
        avail    => urx_avail,
        error    => urx_error,
        clear    => urx_clear,
        rxd      => uart_rxd
    );
```

---

```
uart_tx0: uart_tx
    generic map (
        fullbit => divisor )
    port map (
        clk      => clk,
        reset    => reset,
        din      => txbuf,
        wr       => utx_wr,
        busy     => utx_busy,
        txd      => uart_txd
    );
```



```
rxdata    <= rxbuf;
rx_avail  <= rxbuf_full and not rd;
rx_full   <= rxbuf_full and urx_avail and not rd;
rx_error  <= urx_error;

tx_busy   <= utx_busy or txbuf_full or wr;
tx_avail  <= not txbuf_full;

end rtl;
```

## VHDL (uart\_tx.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
-----
-- UART Transmitter -----
entity uart_tx is
    generic (
        fullbit    : integer );
    port (
        clk        : in  std_logic;
        reset      : in  std_logic;
        din        : in  std_logic_vector(7 downto 0);
        wr         : in  std_logic;
        busy       : out std_logic;
        txd        : out std_logic );
end uart_tx;
```

```
architecture rtl of uart_tx is
```

```
constant halfbit : integer := fullbit / 2;
```

```
signal bitcount : integer range 0 to 10;
```

```
signal count : integer range 0 to fullbit;
```

```
signal shiftreg : std_logic_vector(7 downto 0);
```

```
begin
```

```
proc: process (clk, reset)
```

```
begin
```

```
    if reset='1' then
```

```
        count <= 0;
```

```
        bitcount <= 0;
```

```
        busy <= '0';
```

```
        txd <= '1';
```

```
    elsif clk'event and clk='1' then
```

```
        if count/=0 then
```

```
            count <= count - 1;
```

```
        else
```

```
            if bitcount=0 then
```

```
                busy <= '0';
```

```
                if wr='1' then -- START BIT
```

```
                    shiftreg <= din;
```

```
                    busy <= '1';
```

```
                    txd <= '0';
```

```
                    bitcount <= bitcount + 1;
```

```
                    count <= fullbit;
```

```
                end if;
```



```
        elsif bitcount=9 then          -- STOP BIT
            txd          <= '1';
            bitcount    <= 0;
            count       <= fullbit;
        else                      -- DATA BIT
            shiftreg(6 downto 0) <= shiftreg(7 downto 1);
            txd          <= shiftreg(0);
            bitcount    <= bitcount + 1;
            count       <= fullbit;
        end if;
    end if;
end process;
end rtl;
```

## VHDL (uart\_rx)

```
library ieee;
use ieee.std_logic_1164.all;
-----
-- UART Receiver -----
entity uart_rx is
    generic (
        fullbit : integer );
    port (
        clk      : in  std_logic;
        reset    : in  std_logic;
        dout     : out std_logic_vector(7 downto 0);
        avail   : out std_logic;
        error    : out std_logic;
        clear    : in  std_logic;
        rxd      : in  std_logic );
end uart_rx;
-----
-- Implementation -----
architecture rtl of uart_rx is

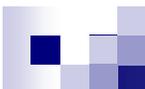
    constant halfbit : integer := fullbit / 2;
    signal bitcount   : integer range 0 to 10;
    signal count      : integer range 0 to FULLBIT;
    signal shiftreg   : std_logic_vector(7 downto 0);
    signal rxd2       : std_logic;

begin
```

```
proc: process (clk, reset) is
begin
    if reset='1' then
        bitcount <= 0;
        count <= 0;
        error <= '0';
        avail <= '0';
    elsif clk'event and clk='1' then

        if clear='1' then
            error <= '0';
            avail <= '0';
        end if;

        if count/=0 then
            count <= count - 1;
        else
            if bitcount=0 then -- wait for startbit
                if rxd2='0' then -- FOUND
                    count <= HALFBIT;
                    bitcount <= bitcount + 1;
                end if;
            end if;
        end if;
    end if;
end process;
```



```
    elsif bitcount=1 then -- sample mid of startbit
        if rxd2='0' then -- OK
            count    <= FULLBIT;
            bitcount <= bitcount + 1;
            shiftreg <= "00000000";
        else -- ERROR
            error    <= '1';
            bitcount <= 0;
        end if;
    elsif bitcount=10 then -- stopbit
        if rxd2='1' then -- OK
            count    <= 0;
            bitcount <= 0;
            dout     <= shiftreg;
            avail    <= '1';
        else -- ERROR
            count    <= FULLBIT;
            bitcount <= 0;
            error    <= '1';
        end if;
    else
        shiftreg(6 downto 0) <= shiftreg(7 downto 1);
        shiftreg(7) <= rxd2;
        count    <= FULLBIT;
        bitcount <= bitcount + 1;
    end if;
end if;
end if;
end process;
```



```
-----  
-- Sync incoming RXD (anti metastable) -----  
syncproc: process(reset, clk) is  
begin  
    if reset='1' then  
        rxd2 <= '1';  
    elsif clk'event and clk='1' then  
        rxd2 <= rxd;  
    end if;  
end process;  
end rtl;
```